ulm university universität ulm

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Wissenschaftliche Arbeit
im Studiengang Lehramt Informatik

Development of an Augmented Reality Component for on the Trail Navigation in Mountainous Regions

vorgelegt von

Lisa Feineis

Juli 2013

Prüfer:	Prof. Dr. Manfred Reichert
Betreuer:	Rüdiger Pryss
Arbeit vorgelegt am:	18. Juli 2013
E-Mail:	lisa.feineis@uni-ulm.de

Abstract

Since the artificial interferences for civilly used GPS devices have been deactivated in 2000, navigating with GPS has become very popular. At first, automotive GPS devices were introduced to help the driver find his or her destination. Thereby, the most common application consists of the user entering his or her destination, whereupon the navigation device calculates the quickest route to reach this place according to predefined conditions. Later, the concept of using GPS for orientation was expanded onto outdoor activities. Here, however, following a preloaded track is considered as prevalent use case. Then, the smartphone allowed integrating GPS based navigation, among many other functions, into a single device. One of the functions most of all smartphones support, is a camera. The combination of camera and GPS sensor opened up the possibility of augmented reality. Thereby, the view upon the real world is supplemented with objects that are displayed as though they were part of reality. In case of a smartphone, virtual objects are placed right onto the camera view. Thus, virtual landmarks can be shown to the user, given their geographic coordinates. In a next step, incoherently showing these landmarks could be extended by showing a whole track, consisting of connected landmarks. This would be a further contribution to facilitate outdoor orientation. In contrast to map based navigation systems, this feature would display a track onto the position where it is actually situated. However, this way of navigation cannot stand alone, but needs to be considered as a supplement to conventional navigation. With the help of an augmented reality based navigation function, the user can compare what he or she reads on a map to what he or she actually encounters in situ. So, he or she can re-evaluate the current situation and make decisions correspondingly. In this thesis, the feasibility of a smartphone application, providing the here described functionality, shall be demonstrated. As an additional feature, information shall be given about which parts of a track are directly visible and which are covered by a geographic structure. Only by this means, the user knows, whether what he or she sees is part of his or her track or if it actually passes behind, e.g., a mountain. To calculate the visibility of a track, an elevation data source needs to be found. So, the approach to find an appropriate source is described in the third chapter of this thesis. Before, the requirements of an application that navigates with the help of augmented reality, are defined. To set the visibility of a track, basic knowledge about geographic calculations and spheric mathematics is needed. After describing these, the design of the finally implemented application is pointed out, and in the next chapter, its implementation is explained on the basis of source code examples. It follows a brief presentation of the application. Then, the results are analyzed concerning their compliance with the predefined requirements. Finally, an outlook is given onto improvements and possible extensions of the here presented application.

Eigenständigkeitserklärung

„Ich erkläre, dass ich die Arbeit selbständig angefertigt und nur die angegebenen Hilfsmittel benutzt habe. Alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken, gegebenenfalls auch elektronischen Medien, entnommen sind, sind von mir durch Angabe der Quelle als Entlehnung kenntlich gemacht. Entlehnungen aus dem Internet sind durch Angabe der Quelle und des Zugriffsdatums sowie dem Ausdruck der ersten Seite belegt; sie liegen zudem für den Zeitraum von 2 Jahren entweder auf einem elektronischen Speichermedium im PDF-Format oder in gedruckter Form vor.“

Ulm, den 18. Juli 2013

Contents

List of Figures

1 Introduction

Twenty years ago the use of computer based services was limited to locally fixed desktop computers and hence influenced the type of software to be used on these computers. Notebooks made it possible to use this software en route, on the train, and on holidays. But only the introduction of the smartphone and tablet computer lead to a fundamental change of software paradigms. By being always and instantly available, smartphone applications offer facilitations in situations where the traditional computer is either not present, or not handy enough. Modern smartphones offer both, a high processing power, and a high-definition display. Thus, even complex software may be run on mobile devices [49] [42] [43] [45], the here presented application being one of them. It is aimed to facilitate orientation on mountain trails by drawing the route directly on the camera display of a smartphone using an augmented reality engine.

1.1. Contribution

This thesis is aimed to develop a component based on an existing augmented reality engine, that extends its functionality to be used for on the trail navigation in mountainous regions. The application runs on the iPhone 4S [29], based on iOS 6.1 [27]. It is meant to draw a chosen track onto the display of the iPhone camera by connecting its track points. Track points that are currently visible, from the user's point of view, should be graphically distinguishable from such that are behind a hill or mountain. Following this, the user can gain information about where exactly his trail is located on the surface of the mountains and hills ahead of him and which part of the trail can be found on the other side. While existing navigation tools may display tracks on maps only and navigate the user alongside them, this application augments the real view of the user by adding the track right onto it. In combination with other outdoor-navigation tools, the application allows the user to compare locations on maps to what he or she actually encounters in reality. He can better evaluate the current situation and make decisions concerning his further way. So, the application could enhance security of mountain activities. From the results of a research on the internet, no application could be found that provides

the described features. For the implementation of the application, data sources are required from which elevation information is gained. With the help of this information, the visibility of a track point can be calculated. To develop a corresponding algorithm, basic geographic and mathematic calculations must be made. Finally, the track must be displayed in an adequate way. Since the augmented reality engine is already capable of displaying single geographic points, these points only need to be adapted to the new requirements and connections lines of the points must be implemented.

1.2. Structure of Thesis

In the beginning of the thesis, the requirements of the application are analyzed. This is done by differentiating realizable features from such that are beyond the scope of this thesis. Afterwards, the handling of data sources is described. So, it is pointed out, which requirements are posed onto a data source to be appropriate for the usage in the augmented reality navigation application. Thereby, the Shuttle Radar Topography Mission (SRTM) [15] data is found to fulfill these requirements. It is stored in Digital Terrain Elevation Data (DTED) [59] files, so this format is analyzed in the next section. Having comprehended the data format, the data can be read and stored in a data base. These preparations are described before finally alternative data sources are presented. To determine the visibility of a track point, knowledge about some geographic and mathematic calculations is needed. This includes the definition of a geodetic datum, to describe a position on the earth's surface. Furthermore, information is needed about which geodetic datum is used on the iPhone and about the accuracy of the therewith gained position values. The algorithm that calculates the visibility of track points is elucidated in the next section. The end of this chapter considers alternative ways of visibility calculation. In the next chapter, the design of the application is presented. This is done by taking a deeper look onto the architecture, the class structure, several communication sequences, and the data persistence of the navigation application. Afterwards, the implementation of the before described features is described with the help of source code examples. Then, screenshots of the running application are explained in the following chapter. Before finally an outlook is given on improvements and extensions of the presented implementation, the application is evaluated according to its compliance with the at first defined requirements.

1.3. Definitions

In this thesis, the *visibility* of track points is often mentioned. This expression describes the fact, whether the user's view onto a track point is obstructed by a geographic structure or not. When referring to a track point or another geolocation that is *in view*, information is given about whether it is within the current view field or not. The latter can, e.g., be the case if the geolocation is situated *behind* the user.

Furthermore, the names of classes are written in italic letters.

2
Requirements

In this thesis, the feasibility of an application that navigates the user by augmented reality in mountainous regions shall be shown. Thereby, a track selected by the user is displayed right onto the camera view of the iPhone. Those parts of the track, that are not visible from the user's point of view, shall be graphically distinguishable from the visible ones. The application is integrated into the existing augmented reality engine AREA [19]. During the implementation, an emphasis is placed on feasibility, performance has been secondary. In the first section, the requirements of the navigation component are listed and described. Then, the scope of the thesis is defined.

2.1. Requirements Analysis Navigation Component

AREA constitutes the base of the navigation component. It is capable of displaying points of interest from the user's surroundings on the camera view of the iPhone [19]. With the help of this functionality, the navigation component shall display the track points of a track selected by the user. Thereby, it should be possible to differentiate between points of interest and track points. Moreover, the order of the track points should be clearly visible. This can, for example, be realized by drawing connection lines in between the adjacent points of a track. Another possibility would be to number the points consecutively. This feature should help to allow the user to trace the further route visually. Tracing will be difficult if the connection lines of nearby points are overlapping or crossing each other. Since the application is meant to be used in mountainous regions, this becomes less frequent as in plain regions. Furthermore, the beginning and the end of a track should be easily to identify. In case of a consecutive numbering of the points, this is given, at least for the first point. The last point, however, needs a special highlighting even then. So, if the end section of the track is without the currently set radius in which geolocations are displayed, the last track point in view may be wrongly taken for the last one of the track. Another special case is given if one of two adjacent points is situated

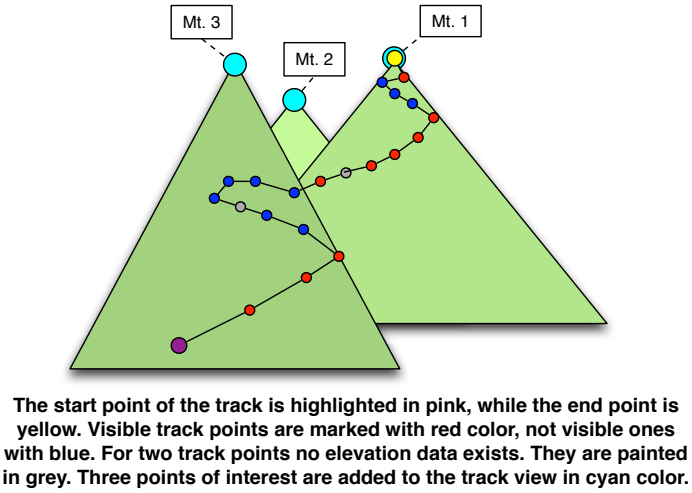


Figure 2.1.: A sketch of the camera overlay view.

within the current view field, while the other one is without. In this case it should be possible to at least identify the direction in which the point, that is out of view, can be found.

It will often be helpful, if the augmented view of a certain track will be enhanced by displaying points of interest. This can, e.g., be the names of surrounding mountains. Thus, it should be possible to add tracks and points of interest to the same view. In contrast to a point of interest, a track point that is currently visible, should be graphically distinguishable from a point that is hidden behind a mountain or a hill. To ensure this feature, an efficient algorithm is needed that bases upon detailed elevation data. Moreover, the result of the algorithm should be as accurate as it is possible without contributions to the algorithm's performance. This is important to ensure a quick update of the track points' visibility as soon as the user's position changes. Thereby, track points for which no elevation data is available should be specially marked to avoid misinformation.

The user should be able to choose which track he wants to have displayed on the camera overlay view. For each track, information about the coverage of elevation data should be given. On this way, the user knows in advance how much he or she can rely on the displayed track. Moreover, it should be possible to display different tracks in one and the same view. Thus, the user can compare routes among each other. Ideally, different tracks are optically distinguishable. The application is meant as a supplement to existing navigation tools (cf. Section *1.1 Contribution*). So, it should be possible to transmit a track that was gained from the internet, a special navigation software, or device to the iPhone to be displayed on the camera overlay view.

Apart from being drawn onto the camera view of the iPhone, the chosen track should also be displayed on the AREA radar view (cf. [19]), if it is within the currently set radius. Following this the user gains information about the direction in which the track is situated and may change his or her viewing direction correspondingly. The upper mentioned algorithm is based on elevation data from the whole surrounding area. A data source needs to be found and processed that provides the application with such data (cf. Section *3.1 Requirements of*

the Data). Once the data is processed, the application needs to establish a connection to a corresponding database and gain its elevation values on that way. Since the application will be designed to be used in regions where mobile internet is rarely available, it should work without data gained from online sources.

Finally, the application should be maintained easily. Therefore, a complete and detailed documentation of source code is indispensable. It should, moreover, be possible to later extend and improve the application's functionality.

A sketch of the camera overlay view of the navigation component may be seen in Figure 2.1. In Table 2.1, all requirements are again listed in a structured way and classified into different types.

2.2. Scope

The navigation component is meant to be applied in mountainous or at least hilly regions. In flat regions, housing and vegetation usually obstruct the view on the area ahead. Moreover, track points do hardly differentiate in height, so themselves or their connection lines, will overlap and the route may not clearly be traced. The user shall, with the help of this application, be able to compare what he or she reads on a map to what he or she actually encounters in situ (cf. Section *1.1 Contribution*). But he or she cannot compare what he or she does not see, which is the case in the flat where one cannot see very far. Hence, there hardly is an utilization for the navigation component in plain regions. While the view in the flat is limited, it is mostly unobstructed onto a vertical geographic structure. Thus, it could be very helpful for climbers to enhance their view of a wall with a preloaded route drawn right onto their smartphones' camera view. This could be realized with the navigation component. The whole visibility algorithm and with it the data source, however, is not necessary in this case. Elevation data will be available as an even grid of elevation points. Thus, in case of a perfect vertical structure, an elevation value will only be given for either a point on top of the wall or on its base. But as long as the user is situated right on the bottom of a wall, he or she will have a full view on it and thus a visibility algorithm is in vain. Only if the wall is partly covered by some different structure, the information about the visibility of a track point on it becomes inaccurate.

As described above, the application differentiates track points that are currently visible from such that are covered by a hill or a mountain. Thereby, the visibility algorithm completely relies on elevation data that has previously been added to the application's sources. The navigation component is not capable of recognizing objects in situ. Thus, a track point may be indicated as visible although it is covered by housing or vegetation. Radar instruments actually record the surface structure of the earth and not its elevation profile (cf. Subsection *3.5.1 The TanDEM-Mission*). On that way, housing and vegetation is already considered. But since this is not a constant condition, such information becomes more useless with the age of the data source.

Another functionality of which the navigation component is not capable of, is drawing a three dimensional model of the surrounding geographic structure right onto the camera view. This could especially be helpful if the weather does not allow a clear view or information shall be gained about the structure of a mountain covered by another one. Nevertheless, by imple-

Table 2.1.: List of Requirements

Requirement	Type
Track points are displayed on the camera overlay view and appear different from points of interest.	functional
Information about the order of track points belonging to a certain track is displayed.	functional
The track is displayed in such way that the further route can be traced clearly.	functional
The beginning and the end of a track is marked.	functional
Even if one track point is within the current view field, but one of its neighbors not, their connection is displayed.	functional
Points of interest can be included in the camera overlay view with the displayed track.	functional
Efficient algorithm to calculate the visibility of a track point.	functional
Correctness of track point visibility.	non-functional
Visible and not visible parts of a track are graphically distinguishable.	functional
Track points for which no elevation data exists are marked.	functional
Quick update of a track point's visibility.	functional
The user can choose from a list of tracks which track he or she wants to be displayed.	functional
Information about the coverage of elevation data is given for all tracks.	functional
Different tracks can be displayed in one camera overlay view and are graphically distinguishable.	functional
Tracks from other sources can be added easily.	implementation
The selected track is displayed on the radar view if it is within the currently set radius.	functional
A data source for elevation data is found and processed.	implementation
Correctness and coverage of elevation data.	non-functional
A connection to the database is established and elevation data is gained on that way.	functional
The application works without internet connection.	non-funcional
Provide maintainability.	non-funcional
Complete documentation of source code.	documentation

menting such a model, calculations would become more complex and the amount of points to be drawn on the screen would increase crucially. This would lead to a significant loss in the application's performance. Moreover, this task is too wide to be treated in this thesis.

Finally, information about tracks can only be gained as a sequence of track points. Thus, the accuracy of the drawn track depends on the interval in which these points have been recorded. The application can only display track points and draw connection lines in between them, but the real analog route will never exactly follow these lines. The larger these intervals become, the more inaccurate is the image of the track on the smartphone's screen. Here, a tradeoff has to be made between performance and accuracy.

3
Data Sources

In this chapter, sources to obtain elevation data are discussed. First, it is elucidated what features the navigation component requires from data sources. In the next section, the finally used data of the SRTM is described in detail. Afterwards, the DTED format which is used to store the SRTM data is analyzed. Then, the way of data preparation is presented. Finally some alternative data sources are pointed out.

3.1. Requirements of the Data

To calculate whether a geographic location is currently visible from the user's point of view, very detailed elevation data from the whole surrounding area is required. Since the iPhone gains its location data via GPS, the position will be given in the geographic coordinate system. Thus, an algorithm that calculates the visibility of a point must request the elevation of a certain geographic location, given by longitude and latitude, and requires a feasible answer. For the achievement of this feature, elevation data should be available as a narrow grid of elevation points covering the surface of the area in question. On the internet, there exist some sources where it is possible to make a request for the elevation of a certain geographic location. So, [20] claims to provide an elevation value for every point on the earth's surface. Furthermore, data is even given in height above mean sea level [20] (cf. Section *4.1 Geodetic Datum*). Since most of these platforms do not name their data sources, it is difficult to evaluate the quality of their data in accuracy and coverage. Moreover, to make a data request, an internet connection is essential. As the application is designed to be used in sparsely populated regions, this cannot be presumed. The reliance on an external internet source, that might arbitrarily be removed, is a further disadvantage. Another idea is to retrieve data from an online map service like Open Street Map [64] or Google Earth [22]. Both services can be used offline and are unlikely to be removed unpredictably. Open Street Maps is based on vector data. It is, particularly, not a topographical map and, thus, not designed to represent elevation data. Nevertheless, there exist some tools to illustrate contour lines and shadings inside a map based on the Open Street

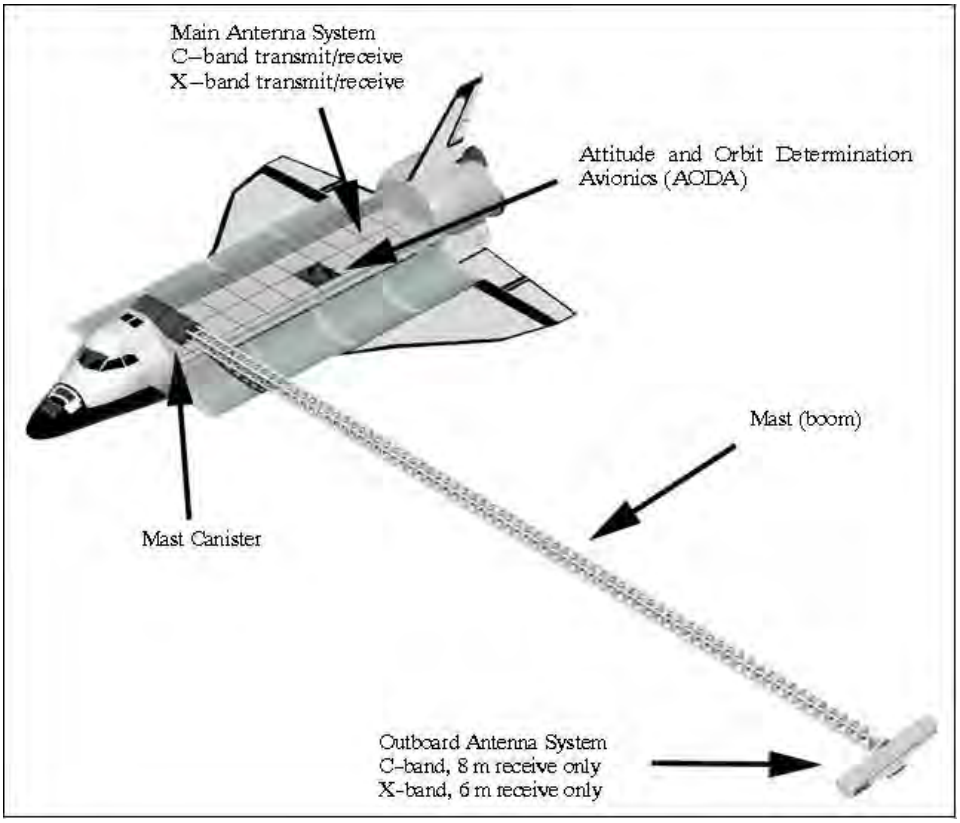


Figure 3.1.: Schematic overview of the SRTM/X-SAR payload with deployed boom (image credit: NASA) [7]

Map Project. These tools utilize SRTM data [64]. To provide Google Earth with a detailed elevation profile, Google offers a plugin that also relies on SRTM data [23]. Therefore, the best solution is to use the SRTM data directly in the navigation component instead of retrieving it via a more or less reliable internet source.

3.2. The SRTM Data

On the Shuttle Radar Topography Mission (SRTM) in 2000, the space shuttle Endeavour was sent on an eleven day long trip around the earth to record a three dimensional model of the earth's surface. This was realized with the help of a 60 meters long mast and two different radar instruments: the German-Italian X-band instrument X-SAR and the U.S. C-band instrument SIR-C [15] [14]. A picture of the prepared space shuttle can be seen in Figure 3.1. While the SIR-C records with a resolution of three arc seconds, the X-SAR has a higher resolution of one arc second. Depending on the latitude, this leads to approximately one elevation point every 90 meters in case of the U.S. records and one per 25 meters for the German ones. Due to the higher resolution of its records, the X-band instrument could only cover 40 percent of the earth's surface, whereas the C-band instrument was capable of recording the whole surface in between 57 degrees South and 60 degrees North [15] [14]. The name of the X-band instruments

Figure 3.2.: Example of the area covered by the records of the X-band radar instrument

derives from the X-shaped records it takes from the surface. No data is available for the area in between the space where the axis of two Xes touch (cf. Figure 3.2). On the 22nd of June 2004, four years after the shuttle was sent off, the obtained X-SAR data was completely processed [14]. It took another six years until the data was made available for scientific purpose and free of charge. After registration, it may be downloaded from Earth Observation on the WEB (EOWEB), an earth observation center platform of the German Aerospace Center (DLR) [15]. The SIR-C data is marketed only by the U.S. Geological Survey (USGS) [14]. As data source of the navigation component, the high resolution X-SAR data was used. Detailedness was considered more important than global coverage. Since the application is designed as a prototype that evaluates a general feasibility, complete coverage is not required. In the section following the next, it will be shown that the X-SAR data does not have many gaps in those areas for which data exists. It will thus be sufficient to test the application in an area which is inside the data's scope. To use the data for the application, it will need to be imported to a *sqlite3* [24] database. To import the data, it has to be extracted from the data source files. The format in which the elevation data of the SRTM is stored is called DTED. In the next section a description of this format will be given.

3.3. The DTED Format

The radar instruments of the SRTM recorded a Digital Elevation Model (DEM) of the earth's surface [58] [14]. To keep this data, different file formats exist, e.g., United States Geological Survey Digital Elevation Model (USGS DEM) [55], Spatial Data Transfer Standard Digital Elevation Model (SDTS DEM) [65], DTED or Digital Image Map (DIMAP) [25] [58]. DTED is the format used for the data acquired on the SRTM [14]. Corresponding to their resolution, the DEMs of the C-band data are coded following the DTED-1 standards while the X-band ones follow the DTED-2 standards. The X-band DTED-2-files can be downloaded in packages covering a geographic area of 10 by 10 degrees. In Figure 3.2, an example of the square described by the 0th and 10th degree of eastern longitude and the 40th and 50th degree of northern latitude can be seen. Each package is, similar to the single files, named after the bottom left coordinate of the 10° by 10° tile. Each package contains

- a picture of the covered area (cf. Figure 3.2)
- a readme file
- a kml-file which offers the possibility to display the covered area on Google Earth
- a directory containing the actual DEM-files
- a directory containing the corresponding Hight Error Maps (HEM).

Whereas the DTED-2-files in the DEM directory assign a longitude / latitude arc second crossing to an elevation value, the files in the HEM directory assign a longitude / latitude arc second crossing to the value 1 or 0, depending on the data coverage of the corresponding geographic location. Each DTED-2-file covers an area of 15 by 15 arc seconds. The file itself is divided into three sections: User Header Label (UHL), Data Set Identification (DSI), Accuracy Descriptor Record (ACC), and the Data Record [53]. The first three sections contain information describing the data, and the last one encompasses data itself. Each section is introduced with a recognition sentinel that makes it possible to locate its information in the file.

3.3.1. The User Header Label

Beginning with the UHL, it contains beside other information:

- the latitude and longitude value of the lower left corner of the data set
- the data interval
- the absolute vertical accuracy and the number of longitude / latitude points / lines [53]

An example of one of the DTED-2-files can be seen in Figure 3.3. In case of this specific DTED-2-file, the latitude and longitude values of the lower left corner are given by *0091500E0483000N*, which means *9° 15' 00" East* and *48° 30' 00" North*. The data interval is described right after with *00100010*. This leads to one arc second for both, the longitude, and latitude interval. Thus, one pixel corresponds to about 25 by 25 meters on the ground. No value (NA) is given to describe the absolute vertical accuracy. Only the readme file of the package specifies the vertical accuracy with $\pm 16m$ absolute and $\pm 6m$ relative. Finally, the number of longitude lines, 901, is given, and the number of latitude points per longitude line, 901.

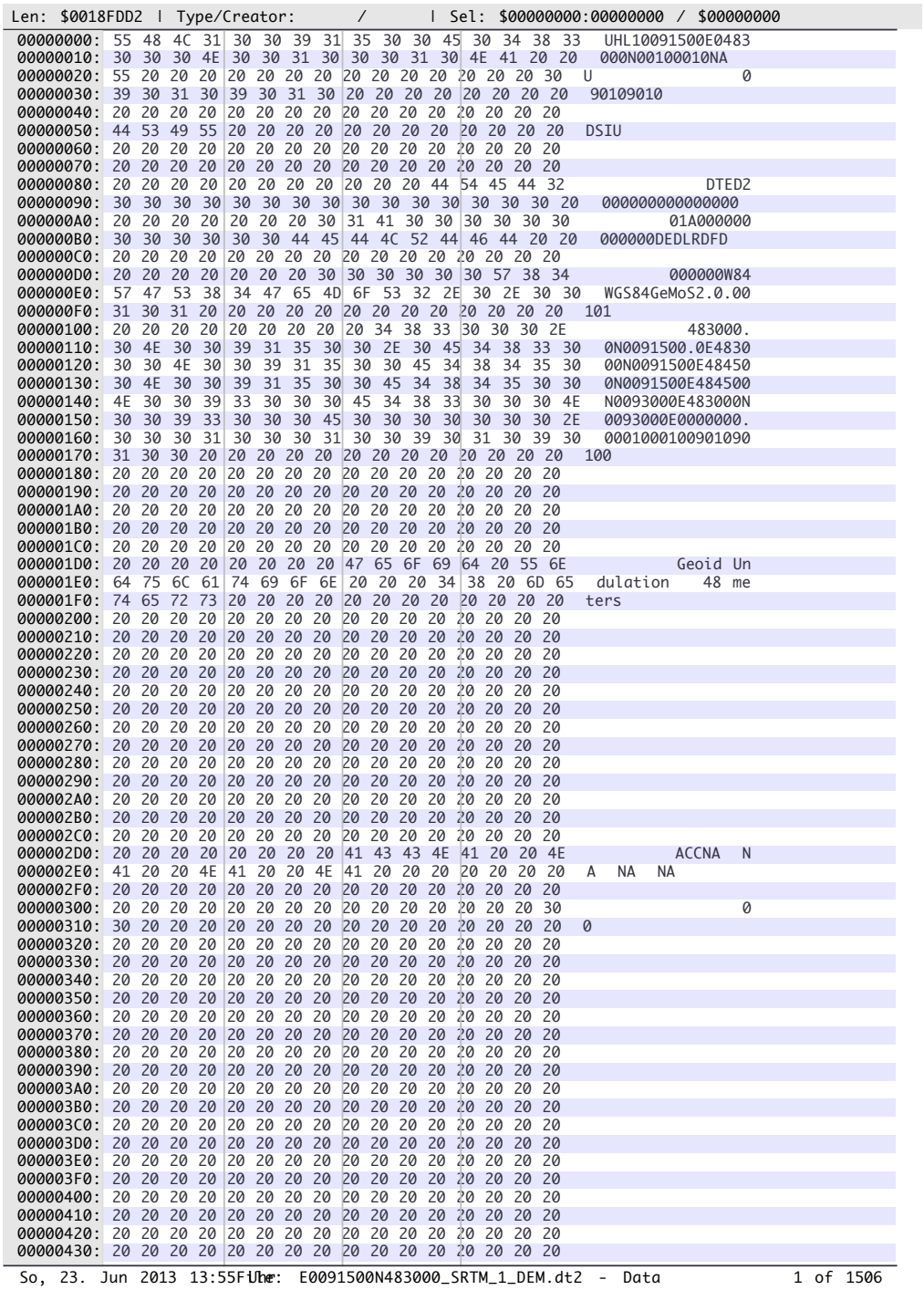


Figure 3.3.: The beginning of a DTED-2-file containing UHL, DSI, and ACC

3.3.2. The Data Set Identification

Next follows the DSI-section. It contains beside other information:

- the product level (DTED-2 or DTED-1)
- the Vertical Datum
- the Horizontal Datum
- more detailed information about the tile's corner coordinates, the intervals, and the number of elevation points
- the tile's data coverage in percent
- the geoid undulation in meters [53]

WGS84 describes both, the Horizontal and the Vertical Datum (cf. Section *4.1 Geodetic Datum*). The coverage of the pictured file is described with *00*, which means 100 %, and the geoid undulation is 48 meters.

3.3.3. The Accuracy Descriptor Record

In the next file section accuracy information is given following the ACC recognition sentinel. Concerning the example file, four times NA is found in the according place. This means, no accuracy data is available for this tile. Again, the readme file provides more detailed information. So, according to this source, the elevation values are provided at a resolution of one meter. Horizontal accuracy is $\pm 20m$ absolute and $\pm 15m$ relative. Although the readme file claims that precise information could be found in the ACC, browsing through the individual files did not extract a single file containing concrete ACC information.

3.3.4. The Data Record

The body of the file is used to store the actual elevation data. According to the DTED format specification of the DLR, every data record begins with an 8-byte preamble containing general information describing the data file [53]. Since the information describing the data file has already been given in the first three sections and it would not make sense to place the same information in each data record, only information describing the individual data record can be meant in this context. The specification describes the first eight bytes following the recognition sentinel as data block count, three bytes; longitude count, two bytes; and latitude count, two bytes. Thereafter follow the elevation values of the latitude point 0 to 900 along the longitude line described by the preceding longitude count value. Finally, a checksum closes the data record block. It its described as an integer summation of 8-bit values of the contents of the block [53]. Since a DTED-file consists of as many data record blocks as it has longitude lines, the data block count and the longitude count have equal values. The latitude count, being the relative latitude value of the first point on the longitude line, is always zero. The checksum turned out to be the summation of 2-byte values of the elevation data section of the data record. A *Perl* [3] script was written to read the elevation data of a single DEM-file and store it into a corresponding database table. Selecting values from the so created database table and comparing them to other elevations of the same geographic location, confirmed the given

interpretation of the data records' structure. The elevation values used for the comparison were acquired from bench marks and elevation requests on internet pages (cf. Section *3.1 Requirements of the Data*). The following gives a detailed description of how data has been read from the file.

3.4. Data Preparation

For testing the application, not all data of the SRTM is required. Even a single tile of 15' by 15' will suffice for testing, as long as its data coverage is complete and the track is completely covered by the data (cf. Section *3.2 The SRTM Data*). The application will be able to load all tiles into memory that are needed for displaying a certain track. Thus, it will be possible to show a track even if it passes through different tiles (cf. Chapter *6 Implementation*). This chapter will describe how a single tile is read and stored into a database using a Perl script. If the application shall be used in a different or larger area, the script can be adopted accordingly. Capturing all elevation data was considered priorly to adding inaccurate values while reading the DEM-files. Relying only on the check sum to test whether the preceding file section is really a data record, led to merely errorless longitude lines being accepted. But most of the single inaccurate points are rejected anyway, when tested if they have an realistic elevation value less than 9000 meters. So, as a different approach to the checksum, the data was checked for the structural characteristics of a data record. While the numeric information in the first three sections of the DEM-file is coded in ASCII, the elevation data is given as hexadecimal values. Thus, conversions are required for these values. All longitude values are given as full arc seconds relative to the lower left corner of the tile that is read in the UHL section of the file. The interval is one arc second. The relative latitude values arise from counting through the elevation points along the longitude line. To get the absolute longitude and latitude values, the corresponding offset from the UHL has to be added. More detailed information about the approach can be found in the comments of Code Listing 3.1.

```
1  # Read recognition sentinel of UHL
2  read(*IN, my $start, 4);
3  # Read longitude value of lower left corner (degrees, minutes, seconds and
       hemisphery)
4  read(*IN, my $lodeg, 3);
5  read(*IN, my $lomin, 2);
6  read(*IN, my $losec,2);
7  read(*IN, my $lohem,1);
8  # Read latitude value of lower left corner (degrees, minutes, seconds and
       hemisphery)
9  read(*IN, my $ladeg, 3);
10 read(*IN, my $lamin, 2);
11 read(*IN, my $lasec,2);
12 read(*IN, my $lahem,1);
13 # Convert read values to integer values in arc seconds
14 my $lo = int($losec) + 60 * (int($lomin) + 60 * int($lodeg));
15 my $la = int($lasec) + 60 * (int($lamin) + 60 * int($ladeg));
16 # 'zaehler' counts through the longitude lines (901 in total), 0 at beginning
17 my $zaehler = 0;
18 # Position in file where the search for the next data record starts
19 my $pos = 0;
```

```
# Read all 901 longitude lines
while($zaehler <= 900){
    # Go to start position of next search
    seek (IN, $pos, 0);
    # Read one byte
    while(my $bytesread = read(*IN, my $buffers, 1)){
        # If the read byte is the recognition sentinel of a data record block, continue reading
        if(ord($buffers) == 170){
            # Read block count and convert it to a decimal number
            read(*IN, my $count1, 1);
            read(*IN, my $count2, 1);
            read(*IN, my $count3, 1);
            my $count = ord($count3)+256*(ord($count2)+256*ord($count1));
            # Check whether this is really the beginning of a new block (read block count value is the same as the value of the counter variable)
            if ($count == $zaehler){
                # Read the relative longitude value (longitude count) of the longitude line and convert it to a decimal value
                read(*IN, my $long1,1);
                read(*IN, my $long2,1);
                my $long = ord($long2)+256*ord($long1);
                # Another reassurance, whether this is a data block: the block count has to have the same value as the longitude count
                if($count == $long){
                    # Read the relative latitude value (latitude count) and convert it to a decimal number
                    read(*IN, my $lat1,1);
                    read(*IN, my $lat2,1);
                    my $lat = ord($lat2)+256*ord($lat1);
                    # If absolute longitude values are required in the database, add the longitude offset from the UHL section to the point counter
                    # my $long = ($long + $lo);
                    # Since this is always the relative latitude value of the first point on the longitude line, it has to have the value 0
                    if($lat==0){
                        # Read the elevation value of all 901 latitude points on the longitude line
                        for(my $i = 0; $i <= 900; $i++){
                            # Read single elevation and convert it to a decimal value
                            read(*IN, my $alt1,1);
                            read(*IN, my $alt2,1);
                            my $alt = ord($alt2)+256*ord($alt1);
                            # If absolute latitude values are required in the database, add the latitude offset from the UHL section
                            #my $lati = ($i + $la);
                            # If this is a realistic value on the surface of the earth (65535 in hex: ffff means no value available)...
                            if($alt<9000){
                                # ...Insert the latitude and longitude value into the database
                                $dbh->{AutoCommit} = 0;
                                $dbh->do("INSERT INTO E915N4830 (long, lat, alt) VALUES ('$long', '$i', '$alt');");
                                if ($dbh->err()) { die "$DBI::errstr\n"; }
                                $dbh->commit();
```

```
64                                  }
65                              }
66                          }
67                      }
68                      # Start next search after the last read byte
69                      $pos = tell IN;
70                      # Increase the count variable to read the next longitude line
71                      $zaehler = $zaehler+1;
72                  }
73              }
74          }
75          # Sometimes a longitude line is missing. In this case the search ends at
            the end of the file. To skip the missing line, start looking for the next
            one.
76          $zaehler = $zaehler+1;
77      }
```

Listing 3.1: Perl Script used for reading a single tile and storing it into a database table.

3.5. Alternative Data Sources

Although the SRTM data proved to be fitting as data source for the navigation component, it has some disadvantages which other sources do not have. In 2013, the successor mission of the SRTM will make its first data sets available. This data will cover the whole globe. The probably most promising approach is, at least for local data, to acquire data from the land surveying offices in Germany. This data is used to produce topographical maps and hence considers heights above sea level. In the following section these alternatives are discussed.

3.5.1. The TanDEM-Mission

In 2011, Prof. Dr. Stefan Dech, director of the German Remote Sensing Data Center, claimed that the SRTM data's accuracy had not yet been surpassed [14]. Currently, the X-SAR instrument that was used to record the elevation data on the SRTM is in service for a different mission. Since June 2010, the national mission TanDEM-X of the German Aerospace Center controls two almost similar satellites carrying each a X-SAR instrument to record a complete and more detailed model of the earth's surface [16]. By synchronizing two different X-band radar instruments, the characteristic data gaps of this instrument can be avoided. What concerns the vertical resolution of the thereby acquired data, it will be with two meters the double value of the SRTM data. The horizontal resolution, in contrast, will be more detailed. So, data intervals will have half an arc second length, which means a bisection of the SRTM value [16]. Whether this new data source would improve the navigation component's accuracy, is difficult to predict. Accuracy data is not yet available of this mission's records. Also it is difficult to say how horizontal and vertical resolution and accuracy should be weighted. But the single fact that the new data will cover the whole earth's surface would fundamentally improve the applications usability. For testing, though, this is not an important issue. The TanDem-Mission will take five years in total, while the SRTM took only eleven days [16] [15]. This remarkable discrepancy is owned to the different bearing objects that were used to carry the radar instruments. So, this year, the first data sets of the TanDEM-Mission are expected [15].

The mission was realized in cooperation with EADS (European Aeronautic Defence and Space Company) Astrium [2], who co-financed the project with 26 Million Euros, approximately 30 percent of the overall costs. In return, Astrium will market the recorded data exclusively [16]. Thus, at least at the beginning, it will be rather unrealistic to obtain the new data affordably. Although being very detailed and global, data acquired via radar instruments has some characteristics that are not always beneficial. First of all, the elevation refers to the earth's surface and not to the actual ground [66]. In particular, this could be an advantage in case of the navigation component, because track points might also be hidden behind trees and houses. On the other hand, vegetation and housing are changing continuously and are thus manipulating the view. Moreover, all elevation data acquired with radar considers a geometrical model of the earth. Weight fields, that form the base to calculate the actual height above sea level, are not considered. Only the geoid undulation given for each DEM-file considers this aspect. But especially in mountainous regions, for which the application is particularly interesting, the geoid undulation varies up to some meters even for the same tile [13] (cf. Section *4.1 Geodetic Datum*). Finally, the SRTM data has some missing pixel values even for the area that was covered by the radar instrument. There are some sources that claim to have filled the missing pixels, but they did either only offer the DTED-1-files, cover only a part that did not fit for testing, or they did not seem reliable. That is why the original data was used for this application. Later on, the gaps did not prove to be very widespread. Nevertheless, there might be some alternative data sources to be used.

3.5.2. Topographic Maps

In Germany, local land surveying offices are responsible to measure the terrain also considering elevation. If no point is set of global coverage, this might probably be the most accurate data to be obtained. It is especially used to produce topographical maps with partly very detailed contour lines. With the introduction of outdoor GPS devices, these maps are also digitally available and are compatible with geographic coordinates. In comparison to the SRTM data, the elevation is given in height above sea level which is a major advantage. Using this data is an option that was not followed up for this application. The SRTM data was easy to obtain and sufficient in detailedness. The data from the land surveying offices will probably not be free of charge. Moreover, it is questionable whether the elevation data can be extracted easily from the used storage format. However, this is a matter that is worth being followed up as a possible future improvement of the application.

4

Geographic and Mathematic Basics

In this chapter, those geographic and mathematic basics are described, that are required to achieve the navigation component's functionality. First, the geodetic datum is explained, including both types, the horizontal and the vertical datum. Next, it is described how information about the geodetic data on the iPhone has been obtained. Then, a comparison of the accuracy of different position values and measurements is made. How the implemented visibility algorithm has been elucidated is described in the next section, before, finally, some alternative ways of visibility calculation are presented.

4.1. Geodetic Datum

"Geodetic datums define the size and shape of the earth and the origin and orientation of the coordinate systems used to map the earth" [5]. Thereby, two different geodetic data can be distingushed. One datum describes the horizontal position on the earth's surface and another one delivers an elevation value.

4.1.1. Horizontal Datum

In case of the horizontal datum, the earth's shape is approximated to a geometric figure, the ellipsoid. A three dimensional ellipsoid is a shape that arises from rotating an ellipse alongside one of its axis [39]. Thus, it is a generalization of a sphere. As coordinate system, the geographic coordinates are widely used. This model splits the earth in two hemispheres, for both dimensions. So, the latitude value is 0° on the equator and increases by approaching the poles to the value 90°. Thereby, it describes the smaller angle between an axis that leads through two facing points on the equator and the axis between the position point and its counterpart on the other side of the earth (cf. Figure 4.1). The latitude value of a point on the northern hemisphere is marked by 'N', while 'S' is the symbol for one in the south. Longitude

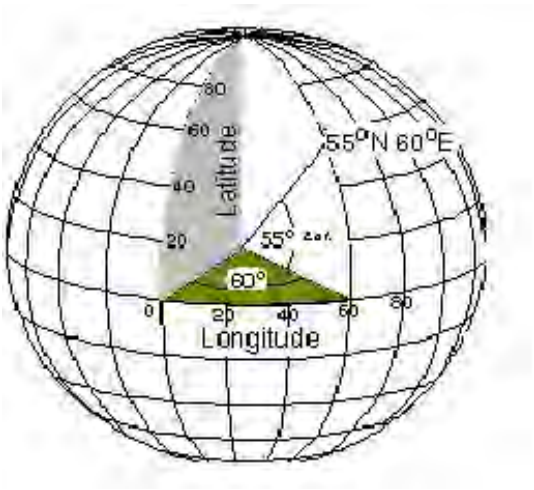


Figure 4.1.: Geographic Coordinates [8]

values are 0° alongside the elliptic arc that starts on one pole, leads through Greenwich in England, and ends on the other pole. The opposite part of this arc has the value 180°. Thereby, again, two hemispheres arise. The eastern one is denoted by the symbol 'E', while the Western is described by 'W' [61] (cf. Figure 4.1). On this model, the earth's center is used as origin of the ellipsoid [39]. While the equator serves as one of the origins of the coordinate system, the origin meridian is chosen arbitrarily. With the help of satellites, it has become possible to develop precise and global applicable ellipsoid models of the earth [57]. The one used by GPS is called WGS84 (World Geodetic System) [39], and a datum with the same name is based on it [39]. The SRTM data, as well, is given in this datum (cf. Section *3.3 The DTED Format*). Those ellipsoid models that were used in the past, try to best adopt the earth's surface in a specific region and not the whole globe [39] (cf. Figure 4.2). Thus, they are not earth centered and also their latitude and longitude values differ from those measured today [39]. On the other hand, they serve much better to describe the real shape of the earth in the region they are used for. On that way, a good ellipsoid model locally almost corresponds with the figure of the earth, that is actually described by a geoid. The latter also serves for the calculation of the elevation [62].

4.1.2. Vertical Datum

Since the ellipsoid is mathematic calculable, it serves best to map a point on the earth's surface onto a coordinate system [39]. What concerns the elevation, the shape of the earth is usually approximated by a geoid to define a vertical datum [60]. In contrast to the ellipsoid, the geoid is a geophysical model (cf. Figure 4.2). So, every point on the earth's surface holds a potential, that arises from a combination of gravitation and centrifugal force which is called the gravity potential of the earth [39]. Alongside the earth's surface there are parallel regions on which this potential is constant. Among them is one region, that corresponds to the average sea level and is imaginarily continued beneath the mainlands [46]. Thus, the geoid model is defined. Through this definition, it is also made clear, that the geoid's surface does not describe the actual *surface of the earth*. Instead, it describes what is called the *earth's figure*. Mostly due

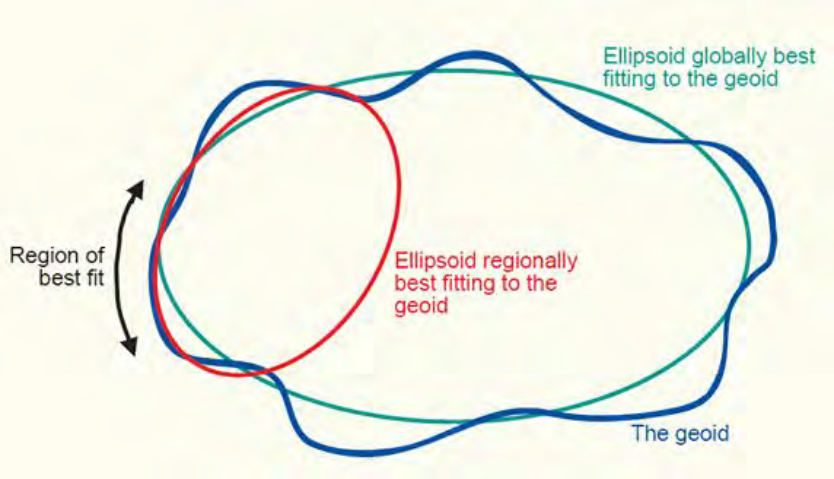


Figure 4.2.: Difference of local and global ellipsoids [33]

to anomalous densities in the earth's mantle, this figure does not correspond to an ellipsoid [62].

While it serves perfectly to describe the earth's figure, the geoid is not the ultimate reference system to calculate the normal height [60]. This value is calculated as the height difference between a point on the earth's surface and the surface of a quasi geoid [47]. The latter derives from the ellipsoid model by subtracting measured height anomalies [48]. The calculation of these values is complex and based on different terrestrial and satellite based measurements [12]. Thereby, the quasi geoid's surface leads through a regional specified water gauge, which's normal height value is postulated as 0 [41]. In Germany, the *German Combined QuasiGeoid 2011 (GCG2011)* [12] is currently in use and refers to the water gauge in Amsterdam [12]. Official elevations are given according to this height system. The difference of geoid and quasi geoid is rather small. So, they are similar on the oceans. In european high-altitude mountains, however, there is a difference of about two meters [48].

The height values that GPS devices gain via satellites, do neither refer to a geoid nor to a quasi geoid model. They are always ellipsoid-heights, which means their elevation values are in relation to the ellipsoid's surface [11], which is the WGS84. The difference between this reference system and the actual normal height can be up to 100 meters and is called *geoidundulation* [60]. Handheld GPS devices, however, come with information about a geoid model in their internal memory [11]. Thus, the current position point is interpolated and a height value is given accordingly [11]. As written above, this value can vary up to two meters compared to normal height. But since GPS height measuring is not very accurate (cf. Section *4.3 Accuracy of Position Values*), this is an acceptable error [18].

4.2. Geodetic data on the iPhone

While handheld GPS devices come with geoid information correcting the GPS height, this is difficult to constitute for smartphones. When developing an application that calculates

elevation data, it is essential to know whether a height correction is already performed by the phone, e.g., the GPS chip, or needs to be implemented manually. Moreover, it would be helpful to know which geoid model is used to perform these corrections, if they are already included. An intensive research on the internet did not lead to any reliable result. There was a posting in a forum, where a user described a problem while implementing an application for Android [21]. The provided method *getAltitude()* seemed to return the ellipsoid-height without geoid correction. Replying to that, another user confirmed that suggestion. He claimed to have developed Android applications himself and, thus, was in knowledge of this matter [51]. For iOS devices, no information at all could be gained about height correction. There was not even any information available about where on the device geoid corrections are performed. If it was implemented on the GPS chip, this would be a matter of hardware, and thus, nothing could be concluded of the phone's operation system. Writing a test application on the iPhone 4S, finally led to the result that the altitude method returned an elevation value that seemed to be corrected by a geoid model. Nevertheless, testing does, in this case, not always give a reliable result. So, the absolute height error is in 95% of the measurements with GPS less than 27.5 m [18] (cf. Section 4.3 Accuracy of Position Values). With a geoid height of approximately 49 meters and a measured value that corresponds to the normal height value, a geoid height correction can be assumed. However, the error is too big to be ultimately sure. Since the elevations of the SRTM data are in relation to the WGS84 ellipsoid (cf. Section *3.3 The DTED Format*), it would be convenient to obtain the GPS elevation data *before* it was corrected. Thus, it would be possible to calculate in the same geodetic datum. So, following request was made to the developer support center of Apple [28]:

> PLATFORM AND VERSION
> iOS
> iphone, iOS 6.0
>
> DESCRIPTION OF PROBLEM
> I am developing an iPhone App in context of my thesis at Ulm University (Germany). Therefore I use a data source that consists of geographic data in WGS84. The altitudes of this data is also in relation to this reference ellipsoid without considering a geoid model (e.g. EGM96).
>
> As far as I know, the altitude data calculated by the iPhone considers a geoid model. Is there any possibility to obtain the GPS raw data from the iPhone, which includes the elevation in relation to the ellipsoid model WGS84? If not, what other solution do you propose?
>
> Thank you for you help!

The answer of the Apple support team:

> Hello Ruediger,
>
> Thank you for contacting Apple developer Technical Supprt (DTS). Our engineers

haver reviewed your request and have concluded that there is no supported way to achieve the desired functionality given the current shipping system configurations.

If you would like for Apple to consider adding support for raw GPS data in future, please submit an enhancement request via the Bug Report tool at <http://bugreport.apple.com>.

While you were initially charged a technical support incident for this support request, we have assigned a replacement incident back to your account.

Thank you for taking the time to file this report. We truly appreciate your help in discovering and isolating issues.

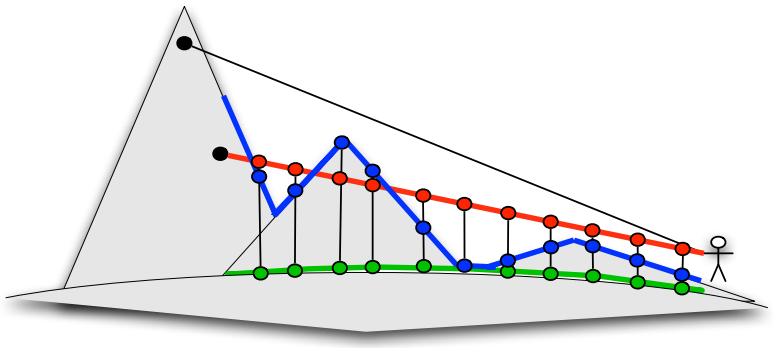
Best Regards,

Developer Technical Support
Apple Worldwide Developer Relations

This, at least, led to the conclusion that the iPhone indeed uses geoid information to correct the height values gained via GPS. Although it would be difficult to implement a back calculation via a geoid model in the navigation component, it would still be interesting to know which geoid model is used. Thus, this question was requested to the Apple support team. This time, Apple's answer was even more deflating. So, they did not even respond to the new question, but merely repeated the impossibility of gaining uncorrected GPS data (cf. Section *A.2 Request about which geoid model is used*). It would have been an option to further research corresponding features for Android devices and then implement the component for this platform. But AREA was, up to then, only available for the iPhone. Thus, the iOS implementation was continued. As GPS data is not very accurate, anyway, it is probably sufficient to calculate with the geoid height correction value that is given individually for each DTED-2-file of the SRTM data (cf. Section *3.3 The DTED Format*). Testing some elevation values of one tile that was situated in the mountains, led to a variation of 1.3 meters in geoid height. In the next section, the accuracy of GPS will be described. This will lead to the conclusion, that such errors do not value crucially.

4.3. Accuracy of Position Values

Before 2000, the GPS signal received by civil devices was artificially interfered by the US Military [18]. This was called *Selective Availability (SA)* [18]. During this time, the absolute error of the obtained horizontal position values was in 95% of the measurements less than 100 meters for civil devices [18]. The accuracy of vertical GPS measurement is generally about 1.5 times worse than horizontal [11]. Thus, the vertical error was between 100 and 300 meters [18]. After the interfering has been switched off, however, accuracy has become much better. Today, the error is less than 21 meters in the horizontal, and less than 27.5 meters in the vertical position [18]. Again, this applies to 95% of the measurements. With the help of barometric altimeters, however, much better values can be achieved. The accuracy of a height



Black points are track points. The upper one is visible, while the lower one is covered by a mountain. The user's view line is drawn in red, while the great circle fraction is green. The blue line considers the elevation alongside the great circle fraction.

Figure 4.3.: Comparison of height values

value measured with a barometric altimeter is about 0.5% of the directly measured height difference [35]. If the height value is determined in multiple steps and calibrated frequently, a high accuracy can be reached [35]. There are some Android smartphones that come with a barometer sensor and can thus determine height values on that way[1]. It was difficult to find reliable information about a similar feature of the iPhone series. According to some posts in forums and comments on online articles (cf. [4], [38] and [50]), there is no iPhone model that comes with a barometric altimeter. Among the sensors of the iPhone 4S, at least, there is no barometer [29]. Because of its high accuracy, a barometer sensor would be a real improvement of the navigation component's functionality. To simply demonstrate the feasibility of such an application, height measurement via GPS will suffice.

4.4. Visibility Algorithm of Track Points

After considering many different ways of how to calculate the visibility of a track point (cf. Section *4.5 Alternative Ways of Calculation*), a rather simple algorithm has been implemented. In consideration of the data interval of the elevation data[2] and the comparable low accuracy of GPS height values, a very accurate calculation would be inappropriate. Moreover, the track points' visibility needs to be updated frequently, consequently, these calculations should not take very long (cf. Section *2.1 Requirements Analysis Navigation Component*). Here, performance is essential to allow the basic functionality of the navigation component and would be hindered by complex calculations. Visibility is calculated for each track point separately. The algorithm compares the height values alongside the user's view line on the track point with the corresponding elevation values gained from the data source (cf. Figure 4.3). Therefore, the longitude and latitude values of the intermediate points in between the user and the track point are needed. The shortest way between two points on the earth's surface is given by a

[1] e.g. Samsung Galaxy S4 [36] and Samsung Galaxy Nexus [4]

[2] This is one arc second, which corresponds to approximately 25 meters on the earth's surface (cf. Section *3.3 The DTED Format*).

section of a great circle [40]. A great circle is an arc that is described by the intersection of a plane, that goes through the center of a sphere and the surface of the sphere [40]. Thus, the radius of the great circle is the same as the sphere's. In case of an ellipsoid, which actually represents the shape of the earth when horizontal positions are determined (cf. Subsection *4.1.1 Horizontal Datum*), extremely complicated expressions are needed to describe a corresponding *great ellipse* [54]. But, for the following calculations, it will suffice to presume a sphere as the earth's shape.

To compute the intermediate points alongside a section of a great circle, the coordinates of the start ($lon_1,\ lat_1$) and the end point ($lon_2,\ lat_2$) need to be converted to radians. Then, the distance of the two points can be calculated:

$$\Delta lon = lon_2 - lon_1 \qquad \Delta lat = lat_2 - lat_1$$

$$a = \sin^2 \frac{\Delta lat}{2} + \cos lat_1 \cos lat_2 \sin^2 \frac{\Delta lon}{2} \tag{4.1}$$

$$\Delta d = 2\text{atan2}(\sqrt{a}, \sqrt{1-a}) \qquad [52] \tag{4.2}$$

in which

$$\text{atan2}(y,x) = \begin{cases} \arctan \frac{y}{x} & x > 0 \\ \arctan \frac{y}{x} + \pi & y \geq 0, x < 0 \\ \arctan \frac{y}{x} - \pi & y < 0, x < 0 \\ +\frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ \text{undefined} & y = 0, x = 0 \end{cases} \qquad [56] \tag{4.3}$$

Here, the haversine formula (cf. Formula 4.1) is used to calculate the distance between two points. In contrast to the *spherical law of cosines* [52], this is numerical well-conditioned even at small distances [52] (cf. also [19]). So, 4.1 calculates the square of half the cord length in between the points and 4.2 gives the angular distance in radians [52]. With 4.3, the *atan2* [56] is defined, that is the arctangent function with two arguments that is used in many computer languages [56]. Since the real distance between two points in meters is never needed, all calculations are made with angular distances. If the distance in meters was needed, Δd would have to be multiplied with the earth's radius. But, with the earth being no perfect sphere, its radius is difficult to be determine unambiguously. Thus, by keeping to the angular distances, calculations gain a higher accuracy.

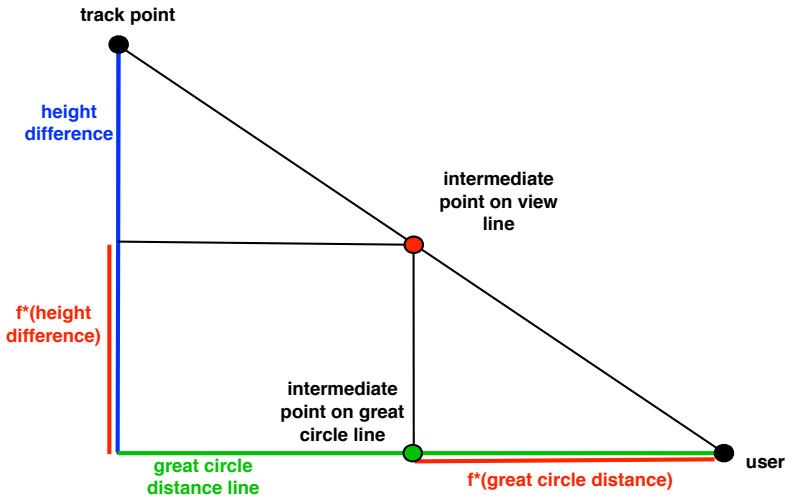


Figure 4.4.: Calculation of an intermediate point's height

Having calculated the distance Δd between the two points, the coordinates of the intermediate points on the great circle section can be computed:

$$A = \frac{\sin\left((1-f)\Delta d\right)}{\sin \Delta d} \tag{4.4}$$

$$B = \frac{\sin\left(f\Delta d\right)}{\sin \Delta d} \tag{4.5}$$

$$x = A\cos lat_1 \cos lon_1 + B\cos lat_2 \cos lon_2 \tag{4.6}$$

$$y = A\cos lat_1 \sin lon_1 + B\cos lat_2 \sin lon_2 \tag{4.7}$$

$$z = A\sin lat_1 + B\sin lat_2 \tag{4.8}$$

$$lat = \operatorname{atan2}(z, \sqrt{x^2 + y^2}) \tag{4.9}$$

$$lon = \operatorname{atan2}(y, x) \qquad [67] \tag{4.10}$$

In which $f \in (0, 1)$ is the fraction of the great circle section for which the intermediate point is calculated. In 4.9 and 4.10 the latitude and longitude values are given for this point. For further processing, they need to be converted to arc seconds. The height (alt_f) of an intermediate point can be computed the following way (cf. Figure 4.4):

$$alt_f = \begin{cases} alt_1 + \Delta alt\ f & alt_1 < alt_2 \\ alt_2 + \Delta alt\ f & \text{else} \end{cases} \tag{4.11}$$

With alt_1 being the altitude of the start point, and alt_2 of the end point. Δalt is the height difference in between the two points. Next, it needs to be determined which intermediate points have to be calculated to be compared with elevation values. To obtain suitable elevation data

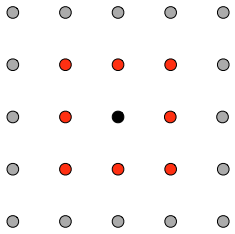


Figure 4.5.: The adjacent points (red) of an elevation data point (black)

values, a corridor is laid around the great circle line that consists of two parallel lines. Both lines have the same distance to the great circle line. Every four points of the elevation data grid describe a square. Thus, the corridor should contain the nearest point of each square it crosses. This is achieved by selecting a corridor width of $\sqrt{2}$ arc seconds[3]. So, it is also assured, that at least one of two adjacent points[4], of which one lies on one, and the other on the other side of the great circle line, is included in the corridor (cf. Figure 4.6). Its elevation value will be directly compared to the height value of an intermediate point. To determine the points that lie in between the corridor, their distance to the great circle line has to be calculated. These calculations can be realized with the help of the (spheric) Pythagoras [32], which can be approximated through the plane Pythagoras for small distances. Thereby, square roots need to be taken, which is a rather expensive calculation. Thus, a good preselection of points should be taken, which are then tested of lying inside the defined corridor. As written above, every four data points form a square. By choosing those points, that are part of the squares the great circle line crosses, a good preselection is given. On that way, all corridor points are included (cf. Figure 4.6). This preselection can easily be realized by determining intermediate points alongside the great circle line in an interval of one arc second and then rounding the latitude and the longitude coordinates up and down to full seconds. So, for every intermediate point, four preselection points are gained. For these points, their distance to the great circle line can be calculated and, with the help of the thereby gained values, the dropped perpendicular foot of each point can be determined (cf. Figure 4.6). These foots are intermediate points of the great circle line which's height can be computed (cf. Figure 4.4 and Formula 4.11) and compared to the corresponding elevation point.

This method will, especially because of the calculations based on Pythagoras, still be quite expensive. So, the corridor points can be approximated by a different, rather simple, method. Here, intermediate points are calculated the same way as for the preselection of the other method. But, now, the longitude and latitude values of the intermediate points are both rounded to the nearest whole second. Thus, each intermediate point is mapped onto a corresponding elevation data point. The thereby gained selection approximates the corridor points well enough (cf. Figure 4.6). Moreover, instead of calculating the dropped perpendicular foot, the height value of the previously calculated intermediate points can be taken for comparison. Thus, a further Pythagoras based calculation can be avoided. The previous intermediate points correspond, in most cases, well enough with the dropped perpendicular foot (cf. Figure 4.6).

[3]The interval of the data grid is one arc second (cf. Section *3.3 The DTED Format*).

[4]If one point is connected with all other points in the grid, the adjacent points are the ones which's connection lines do not cross other points. Thus, every point in a grid has eight adjacent points (cf. Figure 4.6).

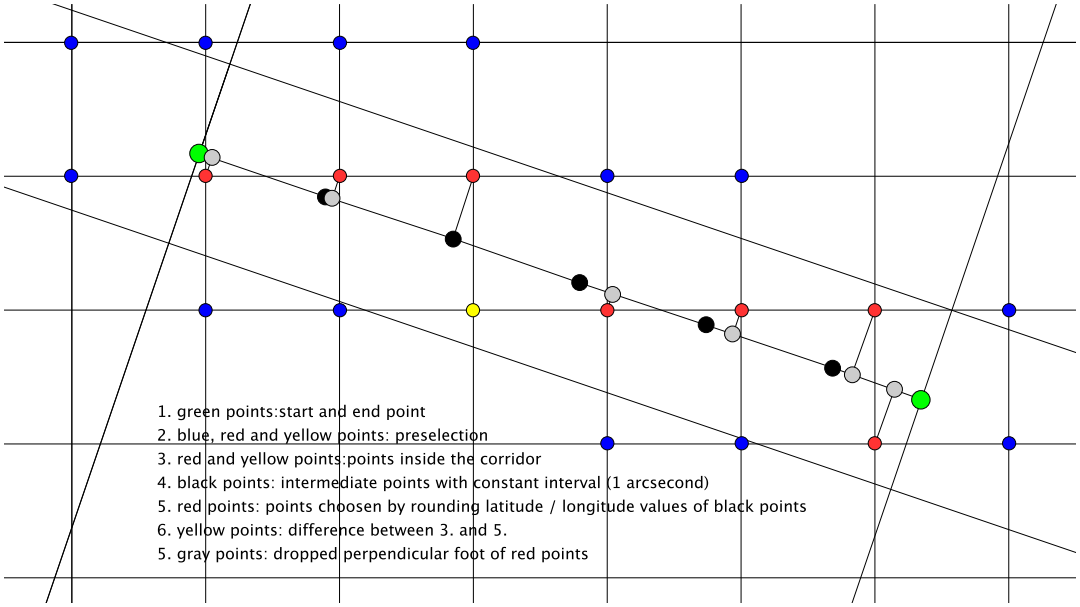


Figure 4.6.: Comparison of different intermediate point selection methods

Having determined which intermediate points of the great circle line are compared to which elevation data points, the actual visibility can be set. If the height values of all intermediate points alongside the view line are higher than their corresponding data points, a track point is set visible. Otherwise, it is not visible (cf. Figure 4.3).

4.5. Alternative Ways of Calculation

An obvious way to calculate a track point's visibility is probably realized by generating a three-dimensional model of the mountain in question. The mountain would then consist of polygons, as it is known of computer graphics. After this, visibility could be determined by checking for crossings between the user's view line and the polygonal planes of the mountain model. This, however, is far too complex to be within the scope of this thesis (cf. Section *2.1 Requirements Analysis Navigation Component*). Moreover, it is questionable whether the computing capacity of a mobile device would suffice to update the mountain model and the track point's visibility with every position change.

The next method does already include the finally implemented idea of comparing intermediate points alongside the user's view line with corresponding elevation data values. Thereby, it would be convenient to calculate with vectors, as it is possible in the plane. However, it is questionable, whether a vector space can at all be defined on the surface of a sphere. In any case, the validity of the vector space axioms would have to be proved. To avoid this problem, geographic coordinates could be converted to three dimensional vectors in a space that origins in the earth's center. This, however, would make calculations rather imprecise. The (spheric) Pythagoras, finally, allows calculations in the geographic coordinate system, which leads to a higher accuracy (cf. Section *4.4 Visibility Algorithm of Track Points*).

The probably most promising method to improve the visibility algorithm is given by interpolating two adjacent points. The elevation points should be chosen in such way, that all data points are included, that have adjacent points (cf. Figure 4.5) on the other side of the great circle line. This is realized by taking the same selection of elevation points as the preselection of the first method described in Section *4.4 Visibility Algorithm of Track Points*. Then, the fraction of the connection line between two adjacent points in which the great circle line crosses is determined. With the help of this value, a height value that lies in between the elevation values of the two adjacent points can be calculated. These calculations are definitely more expensive than the implemented algorithm's. Nevertheless, they would allow a high accuracy.

5 Design

In this chapter, the design of the navigation component is described. Thereby, an especial attention is given to how this application is integrated into the augmented reality engine AREA. In the first section, the general architecture and its individual layers are explained. Next, an overview on the different classes and their relationship is given in the Section *5.2 Class Structure*. Then, crucial communication sequences are described. Finally, the data strategy is pointed out. This includes data gained from files in the GPS Exchange Format (GPX) [10], and the SRTM data that is loaded from a database. Every design element is underlined with a corresponding figure.

5.1. Architecture Design

The architecture of the AREA navigation component follows the Model-View-Controller [63] pattern. By strictly separating between an internal data model and the presentation of data via a graphical user interface (GUI), it allows a high flexibility of software to be reused and, by the way, separates concerns [63]. So it is possible to use one and the same software's model for different GUIs. Though, already AREA has been designed following this pattern [19], the requirements of the navigation component did not allow to take full advantage of this modular architecture. Apart from those classes that had to be added, there were some that had to be modified. Others have been left completely unchanged. But there was no architecture layer that did not have to be adopted. Apart from this, it was very handy that only a reference onto the view controller of AREA was needed to obtain a second camera view with all its functionality. To begin with the model, in this application, it is responsible for managing data and handling XML (Extensible Markup Language) [6] files. There are classes that represent individual objects and such that serve as data storage for the same. Furthermore, a parser which extracts information from GPX-files was added to the existing XML parser. On that way, tracks are loaded into memory. Next, the controller was extended by a class managing the now necessary database connection, and the algorithm described in Section *4.4 Visibility*

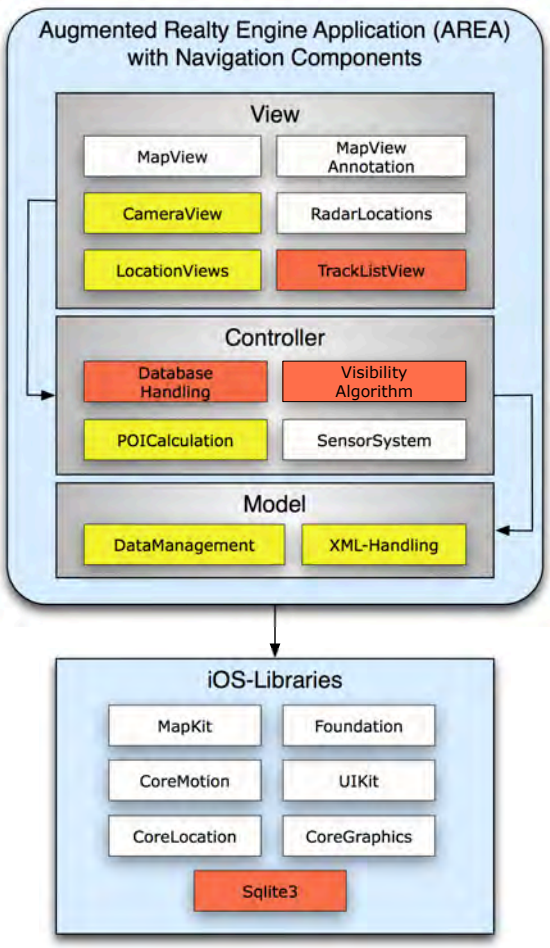


Figure 5.1.: Layered architecture of AREA and navigation components. Based on [19].

Algorithm of Track Points. Finally, the view needed to be adopted to display the connected track points which form a track. Apart from the MVC-Components, the same iOS-Libaries were used as for AREA. One library providing additional database tools was added. These four architectural layers can be seen in Figure 5.1. Thereby, layers on the bottom offer their services via interfaces to such laying above [19]. In the following section the individual classes of the navigation component will be described in more detail.

5.2. Class Structure

In Figure 5.2, the class diagram of AREA with the navigation component are depicted. Classes that had been added newly to the existing engine are painted in red color. Yellow are those which had to be extended to enable the component's functionality. Unmodified classes are left uncolored. As described in the preceding section, changes had to be done in all architectural

layers. Nevertheless, the modular architecture of AREA made it possible that it sufficed to add classes to the model and the view, without modifying the existing ones. In concerns of the controller, those classes had to be extended, that directly communicate with view or model components. The part that is responsible for reading the sensor data could be adopted with no modifications.

Within AREA the controller is responsible for reading sensor data and calculating geolocations [19]. Since the polling of the sensors works the same way for the navigation component as for AREA, the *SensorController* was left unchanged. It transfers data to the *LocationController*. This class calculates whether a geolocation is within the view field and on which place on the screen it needs to be drawn [19]. Corresponding methods were extended by code that adapts the visibility of the track points in view to the new sensor data. For these calculations, *VisibilityAlgorithm* is responsible. *ViewController* receives the computed data from *LocationController*, containing the geolocations which need to be drawn [19][1]. Moreover, it loads the augmented view by accessing the camera view, represented by *CameraOverlayView*, via the *UIImagePickerController* [19]. Thereby, two different instances of the *ViewController* exist. While one is part of the original AREA application and loads the camera overlay view right after displaying the view of the *MapViewController*, the other one is instanced after a track has been chosen from the view of the *TrackListController*. On the camera overlay view, *RadarLocations* and *LocationView* is placed [19]. The latter is a container for the distinctive views of *GeoLocation*. *PointOfInterest* and *TrackPoint* are both subclasses of *GeoLocation*. This causes four different kind of subviews of the *LocationController*: *GeoLocationView*, *PointOfInterestView*, *TrackPointView*, and *ConnectionView*, which represents the connection lines of the track points. The last two were added newly to AREA. To manage the model's data, the controller works on shared instances of different stores. *Store* is the original one that manages all loaded *GeoLocation* objects. Next, *TrackStore* handles *Track* objects that were loaded into memory via *GPXParser*. This second parser was added to the existing *XMLParserForPointsOfInterests*. Finally, *MapStore* manages the elevation data obtained from the database via *DBConnection*. It is stored in instances of *NSMutableDictionary*.

An upside of integrating the navigation component into AREA, has been the object-orientated programming language that allows deriving new classes from existing ones. Thereby, the new classes obtain the facets of the preceding ones. This was the case with the *GeoLocation* and its subclasses. Only through this feature, it was possible that all sensor functionality and most part of the location drawing functionality of AREA could be used without any code changes. The concerned functions were originally designed for *PointOfInterest* objects, but now serve for *TrackPoint* objects as well.

5.3. Communication Sequence

In the following section three communication figures that illustrate crucial sequences of the navigation component's behavior are presented. The first describes how elevation data is loaded into memory, the second points out the user interaction at the track selection process and the third one illustrates a change of the user's position. The parsing of the GPX- files, however, will

[1] Most of the data transfer here is carried out via the implementation of protocols. For more detailed information, see [19].

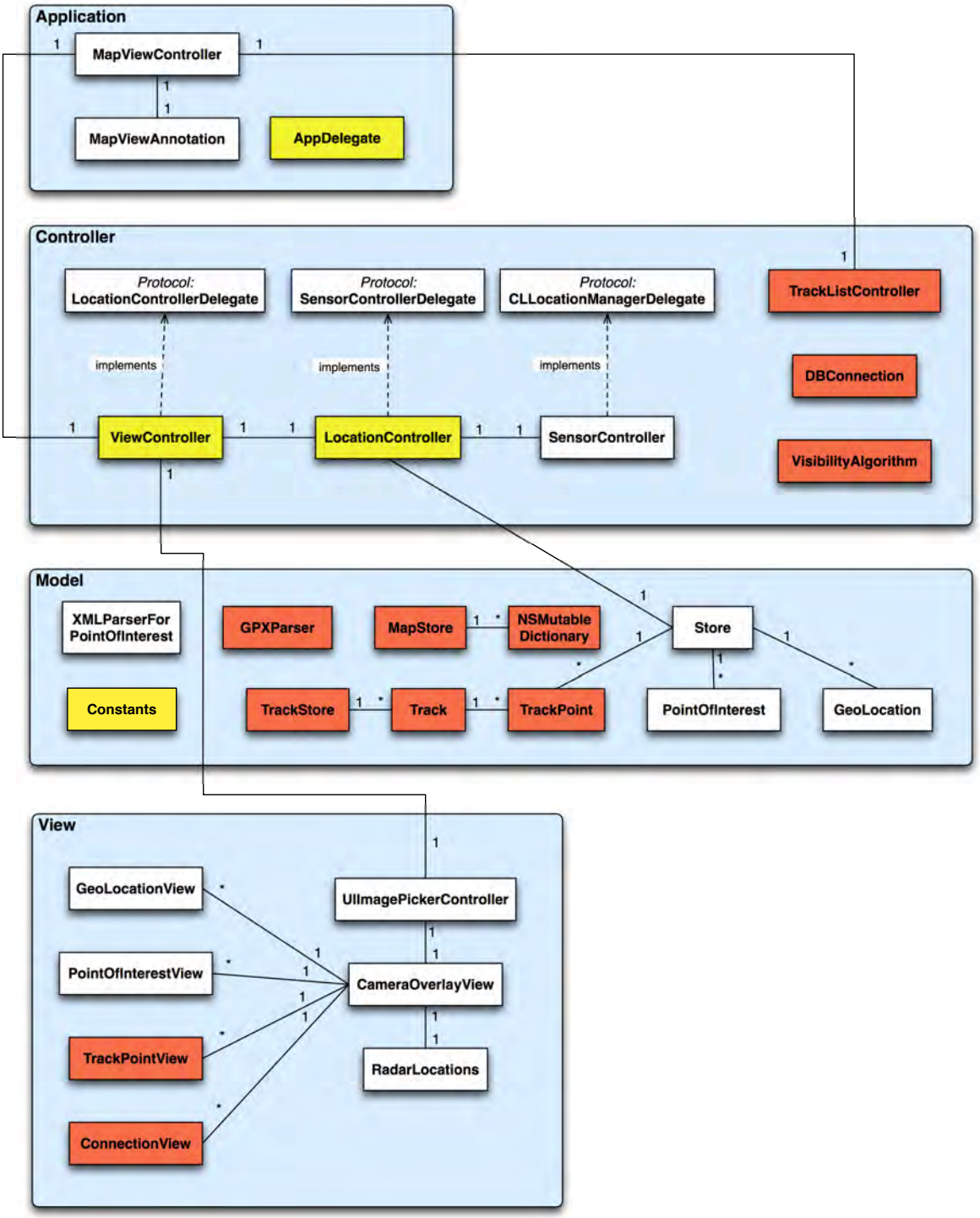


Figure 5.2.: Class diagram of AREA and navigation components. Based on [19].

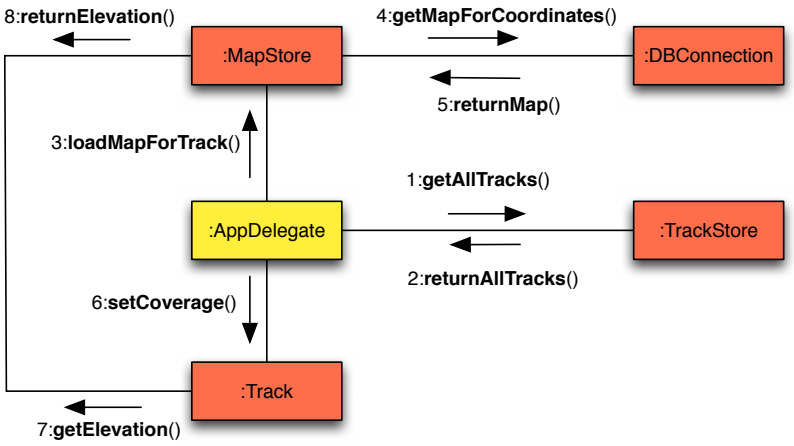


Figure 5.3.: Communication sequence that describes the loading of the geographic elevation data into memory and, for each track, the setting of the coverage value in percent.

be described in Section *5.4 Data Persistence*. Regarding the corresponding figures, elements in red color belong to the navigation component only. Yellow indicates a change of existing AREA source code and white are those parts, that have not been modified.

5.3.1. Loading of Elevation Data

Right after the launch of the application, the *AppDelegate* causes the elevation data to be loaded from the database and the coverage of the tracks to be set (cf. Figure 5.3). In the first step, the *AppDelegate* obtains all loaded tracks by initializing a shared instance of the *TrackStore*. Next, it makes a function call to the *MapStore*, thereby passing the track objects. The *MapStore* extracts the individual *TrackPoint* objects and causes the *DBConnection* to return the corresponding elevation data from the database. The data is then stored in *NSMutableDictionary* objects that are managed by the *MapStore*. In a third call, the *AppDelegate* triggers each *Track* object to determine the percentage of its track points that is covered by the elevation data. This is realized by obtaining elevation data from a shared instance of the *MapStore*.

5.3.2. Track Selection

After the initial loading process, the view of the *MapController* is displayed. By pushing a button on that screen, the *TrackListController* is instanceiated that now manages the track selection (cf. Figure 5.4). This is started by obtaining all loaded *Track* objects in form of a shared instance from the *TrackStore*. Their names and their coverage is then displayed on the view of the *TrackListController*. The user may now choose a track to be displayed on the camera overlay view. The *TrackListController* receives the information about the chosen track and calls a method of the *Store*, thereby passing all *TrackPoint* objects of the track individually. This method is the same that is used in AREA to add *PointOfInterest* objects to the *Store*, which is one of the classes that has not been edited. Afterwards, the *TrackListController*

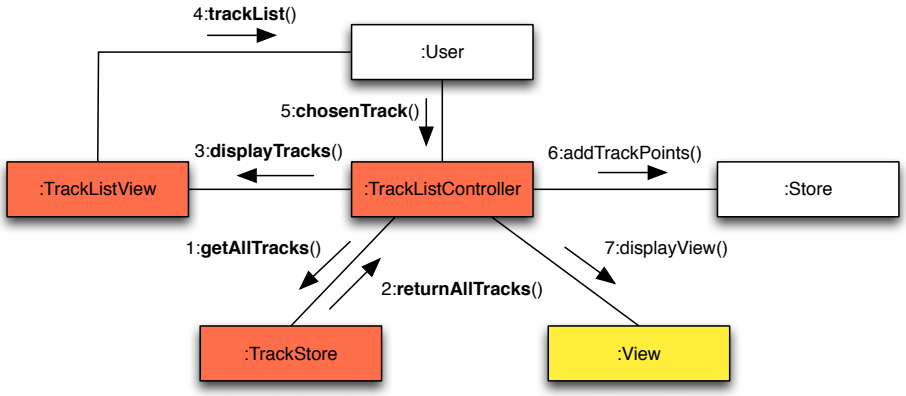


Figure 5.4.: This communication diagram illustrates how a track is selected from a list.

causes the camera overlay view to be displayed. In the next subsection, it will be described how the graphical representations of the track is drawn on this screen.

5.3.3. Change of Position

The initialization of the camera overlay view is similar to a change of the device's position (cf. Figure 5.5). Thus, these two cases can be treated equally. The *SensorController* is the AREA component that recognizes a change of position. By calling a method of *LocationController*, it passes the *CLLocation* object of the new position to the *LocationController*. This class then obtains all *GeoLocation* objects, point of interests and track points alike, that have been stored in the shared instance of the *Store*. Until here, the procedure is similar to the one of AREA. But, in contrast to *PointOfInterest* objects, the view of *TrackPoint* objects depends on information about their visibility. To obtain this information, the *LocationController* passes each *TrackPoint* object that has been found in the shared instance of the *Store* and, moreover, is within a given radius, to the *VisibilityAlgorithm*. This class calculates the visibility of the *TrackPoint* object with the help of elevation data received from the shared instance of *MapStore*. Having computed the visibility, the *VisibilityAlgorithm* causes the *TrackPoint* object to set its properties correspondingly. The remaining sequence is, again, similar to the AREA application. The *LocationController* calculates all surrounding *GeoLocation* objects and caches them together with the calculated *TrackPoint* objects for later use [19]. Then, the *LocationController* passes these *GeoLocation* objects to the *ViewController* that causes the camera overlay view to (re)draw the radar, containing the new surrounding geolocations.

Above only the (re)drawing of the radar view is described. The update of the actual camera overlay view depends on data received via the acceleration sensor and the compass. While the first is necessary when the user's position changes, the latter depends on the device's inclination and orientation. Only when the acceleration sensor and the compass receive new data, the geolocations on the camera overlay view are redrawn. Since the visibility of a track point depends merely on the GPS sensor receiving position data, the code to update this value is put in the corresponding method of the *LocationController*. When the *LocationController* calculates the new positions of the geolocations on the view field, it accesses the cached geolocations

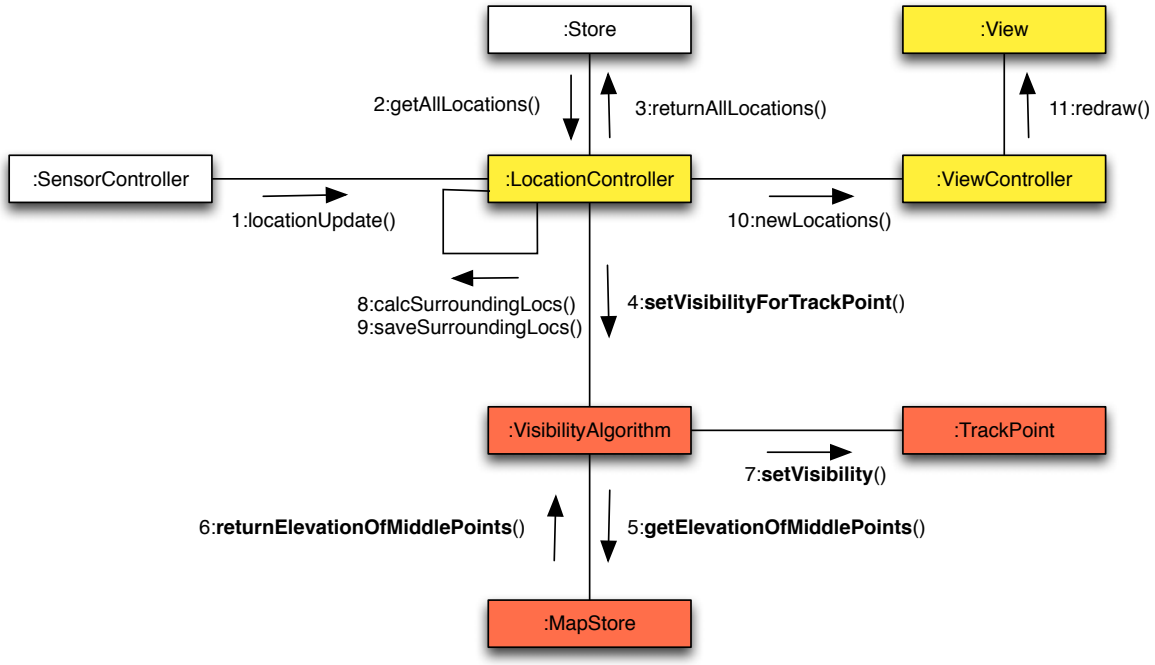


Figure 5.5.: Communication diagram after the user has changed his or her position. Based on [19].

of the position update. The update caused by new compass and acceleration sensor values is completely equal to the procedure of AREA. Its communication sequence can be seen in [19, p. 26].

5.4. Data Persistence

While AREA consists of *PointOfInterest* objects as main data holding entities, the navigation component also obtains data from a database which's information needs to be managed. Moreover, the properties of the former *PointOfInterest* class have been split among the *GeoLocation* class and a new *PointOfInterest* class that now derives from the other one[2]. In addition, another class was derived to represent track points. It holds its own, specific properties and is assembled in a *Track* object, that is, again holding information for itself. This new complexity is pictured by a Entity-Relationship-Model [31] in Figure 5.6. It also shows a diagram of the elevation data obtained from the database, and hold in the *MapStore*. A description of the database can be found in Subsection *5.4.2 Data Base*. Elements of the diagram which's names are written in bold letters are part of the navigation component only.

To begin with, *Store* consists of all *GeoLocation* objects that will be displayed on the camera overlay view, as soon as the user is situated within a certain radius. They contain each a *CLLocation* object, a *DrawPoint*, and are holding the AREA specific information *Distance*,

[2]This is a change that has been performed after the handing in of [19]. Thus, these adoptions cannot be found in the here cited copy.

Horizontal Heading, and *Vertical Heading*. While these properties are calculated during the application's runtime, the *CLLocation* properties *Altitude*, *Longitude*, and *Latitude* must be known in advance. This information is stored in XML-files. In case of the *TrackPoint* objects, a GPX format is used. The *DrawPoint's* properties *yCoordinate* and *xCoordinate* are calculated each time the sensor receives new values. A *GeoLocation* can either be a *PointOfInterest* or a *TrackPoint*. The first contains a *Name* and *Information* as type specific properties. Both must be available in the XML-file. The specific information of the latter, however, will mostly be calculated in runtime. It consists of three booleans:

- Is this the last track point of the track?
- Is it visible from the current point of view?
- Is elevation data available for this *TrackPoint* object?

Each track point, apart from the first one, has a preceding one. A reference on the same will be added to its property list, thereby creating a linked list of *TrackPoint* objects. These objects are assembled in *Track* objects which have a *Name* that is extracted from the corresponding GPX-file and a *cover* value describing the percentage of track points for which elevation data is available. *TrackPoint* objects are stored in the *TrackStore*, of which a shared instance can be obtained.

5.4.1. GPX Parser

> "GPX (the GPS Exchange Format) is a light-wight XML data format for the interchange of GPS data (waypoints, routes, and tracks) between applications and Web services on the Internet. [...] GPX has been the de-facto XML standard for lightweight interchange of GPX data since the initial GPX release in 2002. GPX is being used by dozens of software programs and Web services for GPS data exchange, mapping, and geocachin" [10].

This application shall be a supplement to existing outdoor navigation tools (cf. Chapter *2 Requirements*). Thus, it should be able to display the same track as is shown on, e.g., a handheld navigation device. Furthermore, it should be possible to plan a track on the internet, or with the help of navigation software and then transmit it to a smartphone. By relying on a wide-spread format to be used in the navigation component, the achievement of these requirements can be assured.

```
<xsd:complexType name="trkType">
  <xsd:annotation>
    <xsd:documentation>
      trk represents a track - an ordered list of points describing a path.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string" minOccurs="0">
      <xsd:annotation>
        <xsd:documentation> GPS name of track. </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    (...)
```

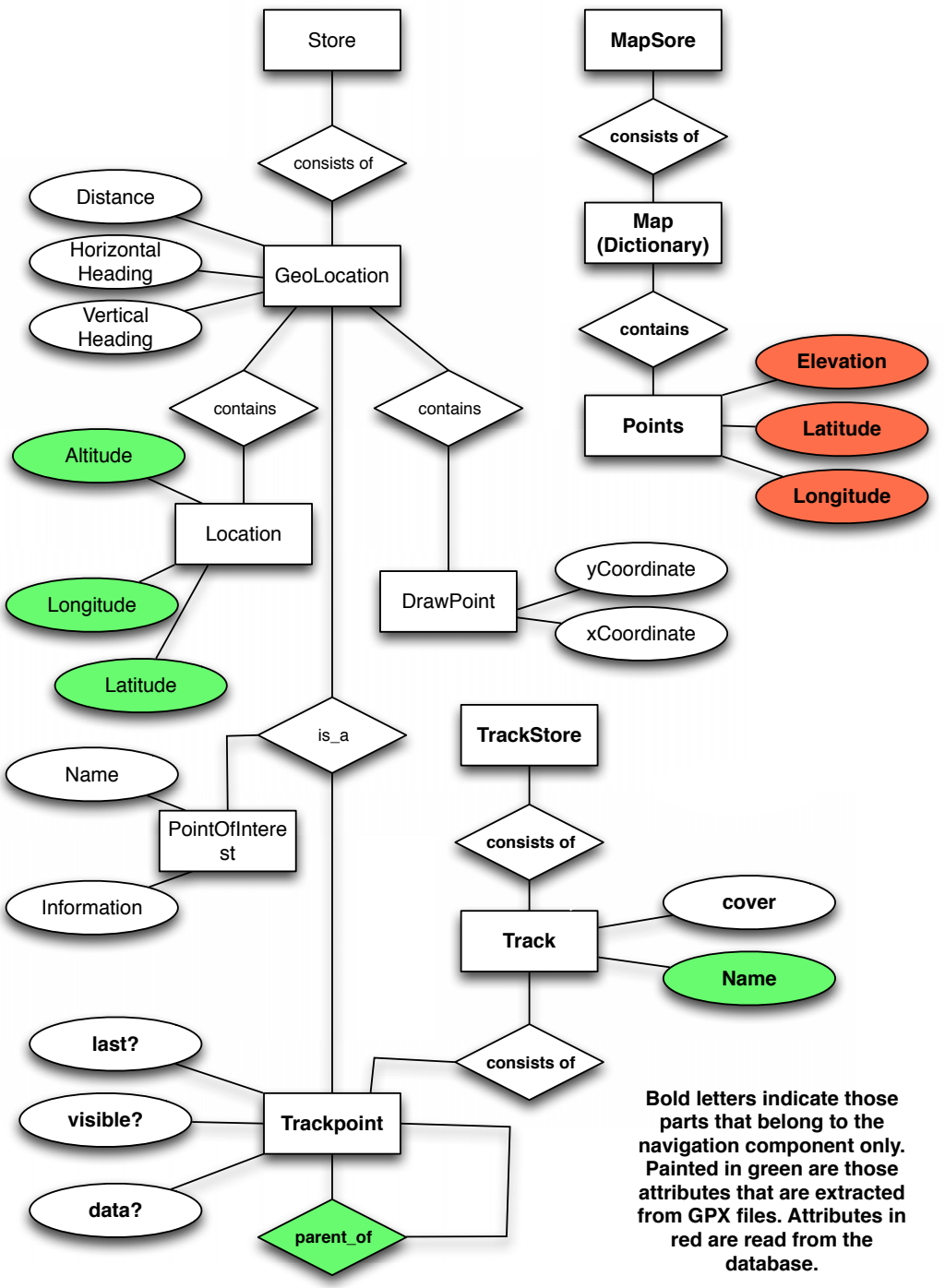


Figure 5.6.: ER diagram of all property bearing entities of the navigation component. Based on [19].

```
14      <xsd:element name="trkseg" type="trksegType" minOccurs="0" maxOccurs="
        unbounded">
15        <xsd:annotation>
16          <xsd:documentation>
17            A Track Segment holds a list of Track Points which are logically
        connected in order. To represent a single GPS track where GPS reception was
        lost, or the GPS receiver was turned off, start a new Track Segment for each
        continuous span of track data.
18          </xsd:documentation>
19        </xsd:annotation>
20      </xsd:element>
21    </xsd:sequence>
22  </xsd:complexType>
23
24  <xsd:complexType name="trksegType">
25    <xsd:annotation>
26      <xsd:documentation>
27        A Track Segment holds a list of Track Points which are logically connected
         in order. To represent a single GPS track where GPS reception was lost, or
        the GPS receiver was turned off, start a new Track Segment for each
        continuous span of track data.
28        </xsd:documentation>
29    </xsd:annotation>
30    <xsd:sequence><!-- elements must appear in this order -->
31      <xsd:element name="trkpt" type="wptType" minOccurs="0" maxOccurs="unbounded"
        >
32        <xsd:annotation>
33          <xsd:documentation>
34            A Track Point holds the coordinates, elevation, timestamp, and
        metadata for a single point in a track.
35          </xsd:documentation>
36        </xsd:annotation>
37      </xsd:element>
38      (...)
39    </xsd:sequence>
40  </xsd:complexType>
41
42  <xsd:complexType name="ptType">
43    <xsd:annotation>
44      <xsd:documentation>
45        A geographic point with optional elevation and time.  Available for use by
         other schemas.
46      </xsd:documentation>
47    </xsd:annotation>
48    <xsd:sequence><!-- elements must appear in this order -->
49      <xsd:element name="ele" type="xsd:decimal" minOccurs="0">
50        <xsd:annotation>
51          <xsd:documentation>
52            The elevation (in meters) of the point.
53          </xsd:documentation>
54        </xsd:annotation>
55      </xsd:element>
56      <xsd:element name="time" type="xsd:dateTime" minOccurs="0">
57        <xsd:annotation>
58          <xsd:documentation>
59            The time that the point was recorded.
60          </xsd:documentation>
```

```
      </xsd:annotation>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="lat" type="latitudeType" use="required">
    <xsd:annotation>
      <xsd:documentation>
        The latitude of the point.  Decimal degrees, WGS84 datum.
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="lon" type="longitudeType" use="required">
    <xsd:annotation>
      <xsd:documentation>
        The latitude of the point.  Decimal degrees, WGS84 datum.
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
</xsd:complexType>
```

Listing 5.1: Type definitions of the *GPX 1.1 Schema* that are relevant for the navigation component [9].

Track files that are added to the application are validated against the *GPX 1.1 Schema* [9]. In the beginning of this section, those properties have been described, that need to be extracted from GPX-files. In Figure 5.6 corresponding attributes are highlighted with green color. Thus, not all information that is given in a GPX-file is needed in the application. Those type definitions of the schema, that are relevant for the navigation component can be seen in Listing 5.1. An example of a GPX-file that follows this is given in Listing 5.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<gpx xmlns="http://www.topografix.com/GPX/1/1" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance" creator="komoot gpx writer" version="1.1" xsi:
    schemaLocation="http://www.topografix.com/GPX/1/1 http://www.topografix.com/
    GPX/1/1/gpx.xsd">

  <metadata>
    <name>Breitenstein</name>
    <desc/>
    <link href="http://www.komoot.de">
      <text>komoot homepage</text>
      <type>text/html</type>
    </link>
  </metadata>

  <trk>
    <name>Breitenstein</name>
    <trkseg>

      <trkpt lat="48.5852557" lon="9.4512613">
        <ele>399.5791015625</ele>
        <time>2013-06-12T11:19:37Z</time>
      </trkpt>

      <trkpt lat="48.5849651" lon="9.4515601">
        <ele>400.0732421875</ele>
```

```
24          <time>2013-06-12T11:20:06Z</time>
25        </trkpt>
26
27        (...)
28
29        <trkpt lat="48.5833924" lon="9.5018125">
30          <ele>782.4619140625</ele>
31          <time>2013-06-12T13:43:02Z</time>
32        </trkpt>
33      </trkseg>
34    </trk>
35  </gpx>
```

Listing 5.2: An example of a GPX-file [34].

5.4.2. Data Base

In Figure 5.6 those properties, that are gained from the SRTM data, are marked in red color. This information is read and stored in a sqlite3 database (cf. Section *3.4 Data Preparation*). The database consists of tables containing elevation data, each covering an area of 15' by 15'. With a data interval of one arc second, this leads to 901 x 901 = 811801 entries per table. The name of a table is compound of a string concatenation holding the following information:

- 'E' or 'W' for eastern or western hemisphere
- longitude value in full degrees
- longitude value in full minutes
- 'N' or 'S' for northern or southern hemisphere
- latitude value in full degrees
- latitude value in full minutes

So the tile covering the area in between 9° 15' to 9° 30' East and 48° 45' to 49° 00' North will have the name *E915N4845*. Each table consists of three columns of *INT* type: 'long' for the relative longitude value in seconds, 'lat' for the relative latitude value in seconds and 'alt' for the elevation in full meters. Thus, the *CREATE* command of the mentioned table would be *CREATE TABLE E915N4845 (long INT,lat INT,alt INT)*. After extracting them from the database, the elevation values of each table are stored into a *NSDictionary*. For the individual tables, relative latitude and longitude values serve as keys. They are concatenated following the same pattern as the table names, only that values are in full seconds and small types are used to describe the hemisphere. A possible value is *e645n234*. These dictionaries are again stored in a dictionary that is managed by *MapStore*. Keys in this dictionary match the names of the tables in the database.

6 Implementation

The navigation component has been implemented with the programming language Objective C [1] based on iOS 6.1 for the iPhone 4S. Xcode [30] has been used as programming environment in the version 4.6.2. In the following sections most of the functionality of the application is explained. Thereby, only those parts are included, that belong to the navigation component. The relationship between AREA parts and the parts of the navigation component are described in Chapter *5 Design*. Explanations of the unchanged constructs of AREA can be seen in [19].

First, data loading is handled by explaining how tracks and elevation data is loaded into the device's main memory. It follows a description of the setting of the coverage value, that gives the user an idea about how much of his or her track is covered by the elevation data. Next, a method is shown, that is triggered when the user chooses a track from a track list. Before connection lines can be drawn onto the camera overlay view, the positions of the geolocations need to be calculated. This is realized by updating geolocations, which is described before the drawing of the connection lines is shown. Thereby, information about the visibility of a track point is required. The implementation of the corresponding algorithm is explained in the final section.

6.1. Loading Track Objects

The *AppDelegate* of AREA was extended by code that initiates a *NSXMLParser*, which then calls its delegate *AREAGPXParser* [26] to parse each GPX-file to a corresponding *Track* object. Thereby, the *AREAGPXParser* extracts the track's information to initiate a *Track* object, set its properties, initiate each of its track points and set each track point's properties, as well (6.1). While doing so, it generates a linked list of track points in such way, that each track point receives a pointer onto its preceding one (cf. Section *5.4 Data Persistence*). This information will later be needed to draw the connection lines of the track points consecutively (cf. Section *6.6 Drawing Connection Lines*). Furthermore, the first and the last track point

will later be highlighted (cf. Section *2.1 Requirements Analysis Navigation Component*). In case of the first track point, no specific property is needed, because it is identifiable by its pointer to the preceding one, which is a null-pointer. For the last track point, however, the property *last* of the *TrackPoint* class is set true (L. 21). This is done for the last read track point as soon as the end tag of the track is read (L. 19). Listing 6.1 shows the setting of a *TrackPoint* object's properties, adding the list of track points to the *Track* object and finally adding the complete *Track* object to the *TrackStore*.

```
1  -(void)parser:(NSXMLParser *)parser didEndElement:(NSString *)elementName
       namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName
2  {
3    ...
4      else if ([elementName isEqualToString:@"trkpt"]) {
5          // Initiate CLLocation property 'location' for current track point
       object with priorly read coordinates and altitude values
6          CLLocation *loc = [[CLLocation alloc] initWithCoordinate:
       CLLocationCoordinate2DMake(_latitude, _longitude) altitude:_altitude
       horizontalAccuracy:0 verticalAccuracy:0 timestamp:[NSDate date]];
7          // set 'location' property for current track point object
8          self.point.location = loc;
9          // set 'parent' property of current track point (= preceding track
       point object)
10         self.point.parent = self.parent;
11         // add current track point object to a list of track points ('points'),
        that is a property of the currently read track
12         [self.points addObject:self.point];
13         self.currentValue = nil;
14         self.parent = nil;
15         // Cache current track point object to be set as parent of the
       succeeding one
16         self.parent = self.point;
17         self.point = nil;
18     }
19     else if ([elementName isEqualToString:@"trk"]) {
20         // This is the end of the track. The last read track point object has
       been cached as parent for the next one and is now marked as last one
21         self.parent.last = true;
22         // Set list of all track points as property of currently read track
23         self.track.points = self.points;
24         // Add current track to track store
25         [self.store addTrack:self.track];
26         self.currentValue = nil;
27         self.parent = nil;
28         self.point = nil;
29         self.track = nil;
30     }
31 }
```

Listing 6.1: Parsing of GPX-files to track objects

After this, the *AppDelegate* calls, for every *Track* object in the *TrackStore*, a method of the *MapStore* that loads for each track all required database tables of the elevation data (cf. Subsection *5.3.1 Loading of Elevation Data*).

6.2. Loading Elevation Data

The *MapStore* is responsible for managing dictionaries in which the elevation data is stored. It is the only class that directly communicates with the *DBConnection*. During the loading of the application, all elevation data maybe needed is once loaded into memory (cf. Subsection *5.3.1 Loading of Elevation Data*). This is realized by checking for each track point, whether a dictionary already exists, which contains the corresponding elevation point of the track point (6.2). The *MapStore* manages a dictionary where all loaded elevation data holding dictionaries are stored. Keys to these dictionaries are given by a string concatenation of the lower left corner of the corresponding data tile (cf. Section *5.4.2 Data Base*). Thus, the key of the corresponding dictionary can be derived of the longitude and latitude values of a track point (L. 9). As soon as a point is found, for which no elevation data dictionary is loaded (L. 14), a method of the *DBConnection* is called (L. 16), passing the name of the missing dictionary and receiving the dictionary object filled with elevation data in return. Once the loading is finished, the *MapStore* will provide other classes with elevation data values. No further database connection is needed. Listing 6.2 demonstrates how each track's point is checked, if its elevation data dictionary is already stored.

```
1  - (void)loadMapForTrack:(AREATrack *)track andCurrentLocation:(CLLocation *)loc
       {
2      // Initiate char array for track name
3      char mapName[11];
4    ...
5      }
6      // All track points of the track are tested, whether they correspond to a
      different dictionary
7      for (AREATrackPoint *point in track.points) {
8          char newMapName[11];
9          getMapforPoint((point.location.coordinate.longitude)*3600.0+0.5, (point
      .location.coordinate.latitude)*3600.0+0.5, newMapName);
10         // Compare dictionary name of previous point to the current one's
11         if (strcmp(newMapName, mapName)) {
12             // if it is in another dictionary, test, whether this dictionary is
       already in the map store
13             NSString *NSmapName = [[NSString alloc] initWithUTF8String:mapName
      ];
14             if ([self.mapStore objectForKey:NSmapName] == nil) {
15         // if it is not, tell database connection to return this dictionary and
       add it to the map store
16                 NSMutableDictionary *map = [AREADBConnection loadMap:mapName
      forLat:latSec andLon:lonSec];
17                 [self.mapStore setObject:map forKey:NSmapName];
18             }
19         }
20         strcpy(mapName, newMapName);
21     }
22     ...
23 }
```

Listing 6.2: Checking for dictionaries to load into the *MapStore*

Given the key of the required dictionary, the class method *loadMap:* of the *DBConnection* returns a dictionary containing the elevation data tile. Thereby, the key of the dictionary corresponds to the name of the database table (cf. Subsection *5.3.1 Loading of Elevation Data*). Listing 6.3 shows the selection of an elevation data tile. After the database connection is established, the dictionary is instantiated (L. 4). Since one elevation tile consists of 901 longitude lines on which 901 latitude points are placed (cf. Section *3.3 The DTED Format*), memory for 811801 dictionary entries needs to be allocated (L. 4). A single row in the database table consists of three column entries:

- the longitude value in arc seconds relative to the lower left corner of the tile
- the latitude value in arc seconds relative to the lower left corner of the tile
- the elevation value of the point given by these coordinates

While stepping through all rows of the elevation data table (L. 12), a string concatenation of the latitude and longitude value is set as key to the elevation value (L. 14,15), and thus inserted into the dictionary (L. 18). Thereby, the characters 'e' and 'n' simply serve as separators between the latitude and longitude values and do neither refer to the eastern nor the northern hemisphere. Finally, the complete dictionary is returned to the *MapStore* (L. 22).

```
1  + (NSMutableDictionary *) loadMap:(char[11])map{
2      ...
3      // Instantiate dictionary to hold 901 x 901 elevation points
4      NSMutableDictionary *points = [NSMutableDictionary dictionaryWithCapacity
       :811801];
5      NSString *NSmapName = [[NSString alloc] initWithUTF8String:map];
6      // Select all values of the database table that corresponds to the
       elevation data tile
7      NSString *query = [@"SELECT * FROM " stringByAppendingString:NSmapName];
8      sqlite3_stmt *statement;
9      int error = sqlite3_prepare_v2(database, [query UTF8String], 1000, &
       statement, nil);
10     if (error == SQLITE_OK) {
11         // Step through database entries
12         while (sqlite3_step(statement) == SQLITE_ROW) {
13             // Set key of elevation point
14             NSString *key = [NSString stringWithFormat:@"e%dn%d",
       sqlite3_column_int(statement, 0),sqlite3_column_int(statement, 1)];
15             // Set elevation point
16             NSNumber *ele = [NSNumber numberWithInt:sqlite3_column_int(
       statement, 2)];
17             // insert key-value pair into dictionary
18             [points setObject:ele forKey:key];
19         }
20     }
21     ...
22     return points;
23 }
```

Listing 6.3: Selecting elevation data of the database

After the elevation data is loaded, the *AppDelegete* calls for each *Track* object a method that sets the coverage of the track and its track points (cf. Subsection *5.3.1 Loading of Elevation Data*).

6.3. Set Coverage

The instance method *setCoverage* (6.4) of the *Track* class calculates the percentage of a track's points for which elevation data exists. Moreover, it sets for each of its *TrackPoint* objects the property *data* (L. 15,18), which is a boolean to describe whether an elevation value exists for the track point or not (cf. Section *5.4 Data Persistence*). This is realized by iterating through the track's points (L. 7) and requesting the corresponding elevation value of the *MapStore* (L. 12). The method *getAltForLon: and Lat:* (L. 12) of the *MapStore* determines at first the key of the dictionary in which the elevation data with the track point's coordinates (rounded to full seconds) is stored. If there is no elevation data available, the *MapStore* returns -1000, otherwise the elevation value obtained from the corresponding dictionary. During the iteration, the number of track points, for which elevation data is available are counted (L. 13). Finally, with the help of this value, the coverage of the track is set in percent (L. 22). So, the coverage value of a track only refers to the coverage of its track points. The surroundings are not considered. The coverage value is meant to give the user information about whether the track he or she has added passes through an area for which no elevation data is available or not (cf. Section *3.2 The SRTM Data*). Since the cover of elevation data points is very good for these areas where data exists, a more detailed calculation of the coverage value is not necessary. The *data* property of the *TrackPoint* object will be needed to mark the corresponding track point on the camera overlay view, if no elevation value is available (cf. Section *2.1 Requirements Analysis Navigation Component*).

```
1  -(void)setCoverage{
2      // Get shared instance of MapStore
3      AREAMapStore *mapStore = [AREAMapStore sharedInstance];
4      // Count number of track points for which elevation data is available
5      int count = 0;
6      // For every track point of the track...
7    for(AREATrackPoint *trkp in _points){
8          // convert coordinates to seconds
9          double lon = (trkp.location.coordinate.longitude)*3600.0;
10         double lat = (trkp.location.coordinate.latitude)*3600.0;
11         // A value of -1000 means that there is no data available for this
       point
12     if([mapStore getAltForLon:lon andLat:lat]>-1000){
13           count++;
14             // Mark: data available
15             trkp.data = true;
16         } else {
17             // Mark: no data available
18           trkp.data = false;
19         }
20     }
21     // Set cover of track point in percent
22     _cover = (int)(((float)count/(float)[_points count])*100.0+0.5);
```

```
23 }
```

Listing 6.4: Setting the coverage of a track

After the application has loaded, the *MapView* is displayed. On its bottom, a button with the name *Choose track* is placed on a toolbar. By pressing that button, a list of all loaded *Track* objects is shown to the user, on which the coverage of the track is displayed right after its name. Thus, the user is informed about a track's coverage value.

6.4. Track List

The *TrackListController* manages a list of all loaded tracks from which the user can choose the one he or she wants to have displayed on the camera overlay view (cf. Subsection *5.3.2 Track Selection*). When the user has chosen a track, the *tableView: didSelectRowAtIndexPath:* method of the *ListController* is called (6.5). Thereby, the index on the *TrackList* corresponds with the *Track* object's index in the *TrackStore*. So, the coresponding *Track* object can be extracted from the *TrackStore* (L. 4). All *TrackPoint* objects of this track are then added to the *Store* (L. 8). The individual views of all *GeoLocation* objects that are stored in this class will be shown as soon as the camera overlay view is displayed on the device's screen (cf. Subsection *6.5 Updating Geolocations*). This happens right after a track has been chosen by the user.

```
1  - (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
       *)indexPath
2  {
3      // The index on the track list corresponds to the track's index in the
       track store
4      AREATrack *track = [[self.trackStore store] objectAtIndex:indexPath.row];
5      //For all points of the track
6      for (AREATrackPoint *point in track.points) {
7          // Add them to the general store of AREA
8        [self.store addPointOfInterest:point];
9      }
10 }
```

Listing 6.5: Selecting a track from a list of tracks

6.5. Updating Geolocations

While the camera overlay view is loaded, those geolocations have to be calculated, which will be displayed on it (cf. *5.3.3 Change of Position*). This procedure is analog to a position update and the *LocationController* is responsible for determining *Geolocations* in view and calculating their heading. Its instance method *didUpdateToLocation:* will be called as soon as the *SensorController* recognizes a change of the user's position. In Listing 6.6 only that part of the method can be seen, which is responsible for updating the visibility of the track points. The rest of the method, and the whole class, is similar to AREA (cf. [19]).

Since only *TrackPoint* objects have to be updated concerning visibility, all *Geolocation* objects in the *Store* are checked of belonging to this class (L. 12). This is done by iterating through all objects in the array that is managed by the *Store* (L. 7). Before, the array needs to be copied, because it cannot be manipulated while being iterated through (L. 5). Moreover, only those *TrackPoint* objects need to be updated, that are within the currently set radius (L. 9,10). During the iteration, the visibility of each track point is set by calling the class method *setVisibilityForTrackPoint: andLocation:* of the *VisibilityAlgorithm* (L. 14). Finally, at the end, the delegate method is called by passing the surrounding locations and the radius (L. 20). This method is part of the *ViewController* which accesses the camera overlay view and draws the geolocation views on it (cf. Section *5.2 Class Structure*).

```
1  -(void)didUpdateToLocation:(CLLocation *)newLocation
2  {
3      ...
4      // The general store needs to be copied, because it cannot be manipulated
       while it is iterated through
5      NSMutableArray *myPoiStore = [NSMutableArray arrayWithArray:_poiStore.store
       ];
6      // Iterate through all geolocations in the general store
7      for (AREAGeoLocation *loc in myPoiStore) {
8          // Only if the point is within the distance, its visibility needs to be
        updated
9          double myDistance = [loc.location distanceFromLocation:newLocation];
10         if(myDistance <= _radius) {
11           // Update visibility if loc is a TrackPoint object
12           if ([loc isMemberOfClass:[AREATrackPoint class]]) {
13               AREATrackPoint *trkp = (AREATrackPoint *)loc;
14         [AREAVisibilityAlgorithm setVisibilityForTrackPoint:trkp andLocation:
       _currentLocation];
15           }
16         }
17     }
18       ...
19     // calculation is finished. So we call the delegate method with the current
        distance, the surrounding locations and the current user location
20     [_delegate didUpdateLocationWithLocations:_surroundingLocations withRadius:
       _radius];
21 }
```

Listing 6.6: Update visibility of geolocations

After the *LocationController* has calculated the new locations that are situated within the radius, they need to be (re)drawn on the camera overlay view. This is realized as soon as the device's heading changes by calling the method *didUpdateHeadingWithLocations:* (6.7) of the *ViewController*. First, three arrays are initialized that contain each one all geolocations that are inside the new view field (L. 5-9). During the first loop, that iterates through all current subviews (L. 11-37), all locations are removed from the first array which had already been inside the previous view field (e.g. L. 29). For them, a (re)draw of their view is not necessary. Instead, an update of their view's frame to a new position on the screen is performed (e.g. L. 30). From the second array, all *TrackPoint* objects are removed, of which a *ConnectionView* already exists (L. 17). This is done for the same reason as for the first array. A separate array is needed, because both, a *TrackPointView* and a *ConnectionView* refer to a *TrackPoint* object.

If a *TrackPoint* object has already been removed because a corresponding *ConnectionView* has been found among the subviews (L. 13), the frame of the *TrackPointView* would not be updated (L. 28-30). The frame of an existing *ConnectionView* is only then updated, if both track points, that are connected by the *ConnectionView*, are also inside the new view field (L. 16). Moreover, the first point of a track does not have a *ConnectionView* to a previous one (L. 16). In case of the other subviews, they are removed, if their corresponding *GeoLocation* object is no longer inside the view field (e.g. L. 33).

The third array is needed to preserve all geolocation objects for the next loop (L. 39-51). That one iterates through the second array, that now contains all *TrackPoint* objects which are inside the new view field, but for which no *ConnectionView* was found among the subviews of the *LocationView*. For each of these *TrackPoint* objects a *ConnectionView* is now initialized (L. 45), if its preceding track point is also inside the new view field (L. 43). Therefore, a list with all locations inside the new view field had to be preserved. Finally, the views of all geolocations, that did not yet have a subview of the *LocationView*, are initialized (e.g. L. 57). These are the ones that are left in the first array. Only the new *ConnectionView* objects have already been added before (L. 46).

```
1  -(void)didUpdateHeadingWithLocations:(NSArray *)locations
2  {
3    // when the heading has changed, the locations on the screen must be redrawn
4    // locations inside the new field of view
5    NSMutableArray *array = [NSMutableArray arrayWithArray:locations];
6    //This field is used to delete all track points of which a Connection to the
       parent already exists
7    NSMutableArray *conArray = [NSMutableArray arrayWithArray:locations];
8    //This field is used to preserve all track points to check them for a parent
       of another track point
9    NSMutableArray *parentArray = [NSMutableArray arrayWithArray:locations];
10   // The location view can contain geo location views, point of interest views,
        track point views and connection views.
11   for(id view in self.locationView.subviews) {
12     ...
13     if([view isMemberOfClass:[AREAConnectionView class]]){
14                 AREAConnectionView *conView = (AREAConnectionView *)view;
15       // If both track points which the ConnectionView connects are inside the
       new view field and the first one of them is not the first one of the track,
        the ConnectionView does not need to be redrawn. An update of its frame
       suffices.
16                 if([array containsObject:conView.trackpoint] && [array
       containsObject:conView.trackpoint.parent] && conView.trackpoint!=nil &&
       conView.trackpoint.parent!=nil) {
17                       [conArray removeObject:conView.trackpoint];
18                       [conView updateFrame];
19                     // Put the connection view behind the track point view
20                       [self.locationView sendSubviewToBack:conView];
21                 } else {
22                       // otherwise remove the subview from its superview
23                   [conView removeFromSuperview];
24                 }
25     } else if ([view isMemberOfClass:[AREATrackPointView class]]) {
26       AREATrackPointView *trkpView = (AREATrackPointView *)view;
```

```
27                  // if the TrackPointView exists also in the array with the
    new locations, just update the frame and move the view to a new position (
    no redraw necessary -> performance)
28                if([array containsObject:trkpView.trackpoint] ) {
29                      [array removeObject:trkpView.trackpoint];
30                      [trkpView updateFrame];
31                } else {
32                  // otherwise remove the subview from its superview
33                      [trkpView removeFromSuperview];
34      }
35      ...
36    }
37  }
38  // ConArray now contains all track points of which no ConnectionView to its
     parent exists, yet.
39  for (id loc in conArray) {
40    if([loc isMemberOfClass:[AREATrackPoint class]]){
41      AREATrackPoint *trkp = (AREATrackPoint *)loc;
42      // ParentArray still contains all locations inside the new view field. If
     the preceding track point of this track point is also within this array...
43      if ([parentArray containsObject:trkp.parent]) {
44        // add a ConnectionView between the two points as subview to the
    superview
45          AREAConnectionView *conView = [[AREAConnectionView alloc]
    initWithLocation:trkp];
46          [self.locationView addSubview:conView];
47          // put it behind the track point view
48          [self.locationView sendSubviewToBack:conView];
49      }
50    }
51  }
52  // the locations (all apart from the ConnectionView) that were not yet
     visible (no subview in locationView) must be initialized and added to the
     locationView
53      for(id loc in array) {
54    ...
55    } else if([loc isMemberOfClass:[AREATrackPoint class]]){
56      AREATrackPoint *trkp = (AREATrackPoint *)loc;
57      AREATrackPointView *view = [[AREATrackPointView alloc] initWithLocation:
    trkp];
58      [self.locationView addSubview:view];
59    }
60            ...
61  }
62}
```

Listing 6.7: Update geolocations

6.6. Drawing Connection Lines

On the camera overlay view, track points are displayed with connection lines in between them (cf. Section *2.1 Requirements Analysis Navigation Component*). Thus, each track point, apart from the first one, is connected to its preceding one. These lines are graphically realized as subviews of the *LocationView*, like the *TrackPointView* and the *PointOfInterestView*, and are

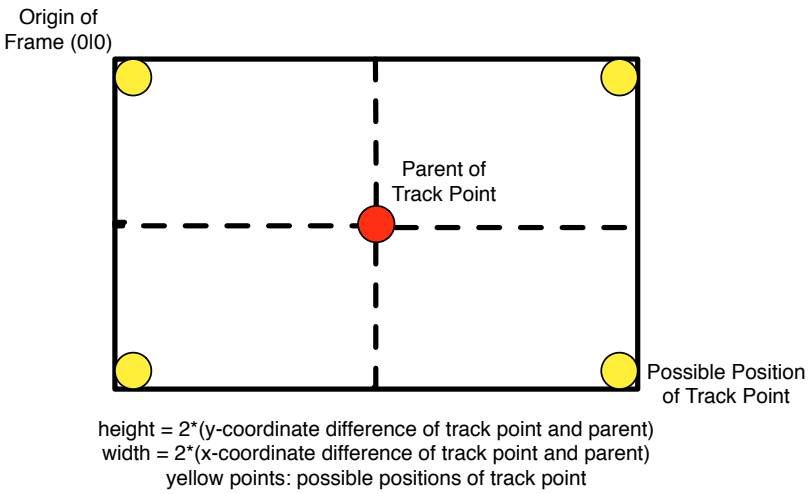


Figure 6.1.: Frame of the *ConnectionView*

defined by the class *ConnectionView*. Each *ConnectionView* object refers to a *TrackPoint* object in its properties. This object, again, refers to another *TrackPoint* object that is its preceding one. Following this, the connection line can be drawn from one point to the other. By being added as another subview to the *LocationView* (cf. Subsection *5.3.3 Change of Position*), it is included in the update routines of the other location views.

The first method (L. 1-9) of Listing 6.8 shows how the frame of the *ConnectionView* object is set, while the second one (L. 11-44) demonstrates how the line itself is drawn onto this frame. Drawing in AREA, and the navigation component, is realized with the help of the library Quartz 2D, which is part of the Core Graphics framework [37]. In this library, the origin of the coordinate system used for drawing is set on the upper left corner. The frame of the *ConnectionView* will be set in such way that the parent of the corresponding track point is placed in its middle and the track point itself is situated in one of its corners (L. 7) (cf. Figure 6.1). Therefore, the height of the frame needs to be chosen double the value of the height difference of the two points (L. 5). The same is done for the width (L. 4).

The *drawRect:* method of an *UIView* is called, whenever a (re)draw of the view object is caused. Here, it determines the relative position of the track point inside the frame and then draws a line from the middle of the frame to this position (L. 36). Thus, the two points are connected. To determine the position of the track point, the x- and y-coordinate values of the two points need to be compared (L. 22,28).

```
1 - (id)initWithLocation:(AREATrackPoint *)trkp
2 {
3     // Calculate height and width of frame
4     self.width = 2*fabsf(trkp.point.x-trkp.parent.point.x);
5     self.height = 2*fabsf(trkp.point.y-trkp.parent.point.y);
6     // Draw frame in such way, that the track point's parent is set in the
      middle, and the track point itself in one of the corners
```

```
7       CGRect frame = CGRectMake(trkp.parent.point.x - self.width/2 - 1, trkp.
        parent.point.y - self.height/2 - 1, self.width + 2, self.height + 2);
8       ...
9   }
10
11  - (void)drawRect:(CGRect)rect
12  {
13      // Calculate width and height of frame
14      self.width = 2*fabsf(_trackpoint.point.x-_trackpoint.parent.point.x);
15      self.height = 2*fabsf(_trackpoint.point.y-_trackpoint.parent.point.y);
16      CGContextRef context = UIGraphicsGetCurrentContext();
17      CGContextSetLineWidth(context, 2.0);
18      // Relative coordinates for track point position on frame
19      float relativX;
20      float relativY;
21      // If the x-coordinate value of the track point's parent is greater than
        the one of the track point, the track point will be set on the left side of
         the frame, otherwise on its right side
22      if (_trackpoint.parent.point.x > _trackpoint.point.x) {
23          relativX = 0;
24      } else {
25          relativX = self.width;
26      }
27      // If the y-coordinate value of the track point's parent is less than the
        one of the track point, the track point will be set on the top of the frame
        , otherwise on its bottom
28      if (_trackpoint.parent.point.y > _trackpoint.point.y) {
29          relativY = 0;
30      } else {
31          relativY = self.height;
32      }
33      // Move to position of parent point
34      CGContextMoveToPoint(context, self.width/2 + 1, self.height/2 + 1);
35      // Draw line to track point
36      CGContextAddLineToPoint(context, relativX + 1, relativY + 1);
37      CGContextStrokePath(context);
38  }
```

Listing 6.8: Drawing the view of the connection line between two track points

The *ConnectionView* should even then be shown on the camera overlay view, if only one of the two points is within the view field, while the other one is without (cf. Section *2.1 Requirements Analysis Navigation Component*). This can be reached by increasing the size of the *LocationView* in such way, that the point, which is outside the view field, will still be drawn onto this view. The size of the AREA *LocationView* is defined in the class *Constants* as 580 x 580 Pixel. To comply with the requirements of the navigation component, it has been increased to a size of 5220 x 5220 Pixel. A dynamic adoption of the *LocationView*'s size is rather difficult to implement. In its final version, the user should be able to set the value of the *LocationView*'s size him- or herself. Connection lines to those points, that are *behind* the user, however, cannot be displayed by increasing the size of the *LocationView*.

6.7. Calculating Visibility

During the update of a geolocation's position (cf. Section *6.5 Updating Geolocations*), the method *setVisibilityForTrackPoint:andLocation:* of the *VisibilityAlgorithm* is called. This causes the method *checkView* (6.9) of the *VisibilityAlgorithm* to determine the track point's visibility following the algorithm described in Section *4.4 Visibility Algorithm of Track Points*.

First, the latitude and longitude values of the user's position and the track point need to be converted to radians (L. 3,4). If the navigation component is used in the western or the southern hemisphere, the latitude and longitude values will have to be multiplied with -1. This case, however, is not covered here. Then, the distance between the two positions has to be calculated with the haversine formula (cf. Formula 4.1) (L. 7) and is also converted to radians (L. 8). Moreover, the height difference of the two points is determined (L. 8). The loop (L. 13-44) follows alongside the great circle line described by the two positions and calculates the intermediate points with an interval of one arc second. This is done by determining the intermediate point's fraction of the great circle line (L. 15) and applying Formula 4.4-4.10. (L. 18-24). Next, the height of the intermediate point alongside the user's view line is calculated following Formula 4.11 (L. 27-32). The elevation of the corresponding elevation data point is obtained by calling a method of the *MapStore*. Thereby, the latitude and longitude values of the intermediate point are rounded to full seconds. Finally, the two height values are compared (L. 38). The height value of the intermediate point has been calculated of the user's elevation and the track point's elevation. The first value is obtained by the device considering a geoid model, while the second one is given by a GPX-file. Both values correspond approximately to the normal height of their position. Thus, the calculation of the intermediate point's height (L. 27-32) can be done without a conversion of the datum. When compared to values gained from the elevation data source, however, a geoid height value has to be subtracted from the elevation data point's value (L. 38). For testing the navigation component, a constant value of 49 meters was subtracted, that corresponds to the geoid height in most parts of south Germany. For the final version, the geoid height that can be gained from the DTED-files (cf. Section *3.3 The DTED Format*) should be stored for each tile in the database and then loaded into the corresponding dictionary of the *MapStore*. If the height values of all intermediate points alongside the user's view line are less then the corresponding elevation data, the track point is visible, otherwise it is set not visible by returning false (L. 40).

```
1  int checkView(double startPoint[3], double targetPoint[3]){
2      // Convert start- and targetPoint to radians
3      double start[3] = {toRadians(startPoint[0]),toRadians(startPoint[1]),
       startPoint[2]};
4      double target[3] = {toRadians(targetPoint[0]),toRadians(targetPoint[1]),
       targetPoint[2]};
5
6      // Calculate distance in radians and height difference between points
7      int distance = getDistance(start, target);
8      int deltaAlt = ab(start[2]-target[2]);
9      double radDistance = toRadians(distance);
10     int i;
11
12     // Go alongside the great circle line and calculate the intermediate points
        with an interval of one arc second
```

```
13      for (i = 0 ; i <= distance; i++) {
14          // Calculate the intermediate point's fraction of the great circle line
15          double f = (double)i/(double)distance;
16
17          // Calculate intermediate point
18          double A = sin((1-f)*radDistance)/sin(radDistance);
19          double B = sin(f*radDistance)/sin(radDistance);
20          double x = A * cos(start[1]) * cos(start[0]) +  B * cos(target[1]) *
    cos(target[0]);
21          double y = A * cos(start[1]) * sin(start[0]) +  B * cos(target[1]) *
    sin(target[0]);
22          double z = A * sin(start[1]) +  B * sin(target[1]);
23          double lat = atan2(z,sqrt(pow(x,2)+pow(y,2)));
24          double lon = atan2(y,x);
25
26          // Calculate the height of the intermediate point alongside the user's
    view line
27          double alt1;
28          if (start[2]>target[2]) {
29              alt1 = f*deltaAlt + target[2];
30          } else{
31            alt1 = f*deltaAlt + start[2];
32          }
33          AREAMapStore *mapStore = [AREAMapStore sharedInstance];
34      // Get the elevation of the corresponding elevation data point
35          int alt2 = [mapStore getAltForLon:toSeconds(lon) andLat:toSeconds(lat)
    ];
36
37        // If the height value of the user's view line is less than the elevation
    , continue testing the other intermediate points. 49 meters = geoid height
    + 6 meters for vegetation height
38          if(alt1 < (alt2-55)){
39          // Otherwise return that the track point is not visible
40              return 0;
41          }
42      }
43      // Return that the track point is visible
44      return 1;
45  }
```

Listing 6.9: Calculate visibility of track points

7
Presentation of the Application

In this chapter, the completely implemented navigation component is presented. Thereby, screenshots of the running application are displayed and explained. In most of the pictures, the view is slightly shifted, because a little shaking of the hand holding the smartphone has been inevitable. With the help of a tripod these complications could be avoided.

After the loading of the application, the AREA map view is displayed on the device's screen (cf. Figure 7.1). Thereon, nearby points of interest are marked. Moreover, a button upon a toolbar at the bottom of the view navigates the user to a list of tracks (cf. Figure 7.2).

The track list displays all loaded tracks, each followed by a percentage that describes how many of the track's points are covered by elevation data values. By choosing one of the tracks, the user is forwarded to the camera overlay view on which the selected track is displayed (cf. Figure 7.3 - 7.6).

In Figure 7.3 the beginning of a track can be seen. Thereby, the first track point is highlighted in violet color. The track passes over the meadow below and then leads right to the user's position. Although the track points are directly visible, they are painted in blue color, which indicates that the point is situated behind a geographic structure. This is probably caused by the flatness of the meadow, which increases the inaccuracy of the visibility calculations. On the top left corner of the view, a small fraction of the track's ending is displayed.

Figure 7.4 shows different segments of the track ahead of the user, that cross the same contour line. While at the first crossing, the path's direction can be identified, the further route becomes rather confusing near the point of interest *Burg Teck*. The track leads right ahead of the user, straight over the meadow. So, the track point near the tree is clearly visible and correctly marked in red color. Afterwards, the track passes behind a hill and thus, the next track point is not visible. The past next one, however, is visible, because the track emerges again from behind the hill. Thereby, the camera overlay view is slightly shifted to the bottom-right. This can also be seen at the point of interest *Burg Teck* in the left edge of the view. Furthermore,

Figure 7.1.: AREA map view with 'Choose track' button

Figure 7.2.: List of loaded tracks with coverage value

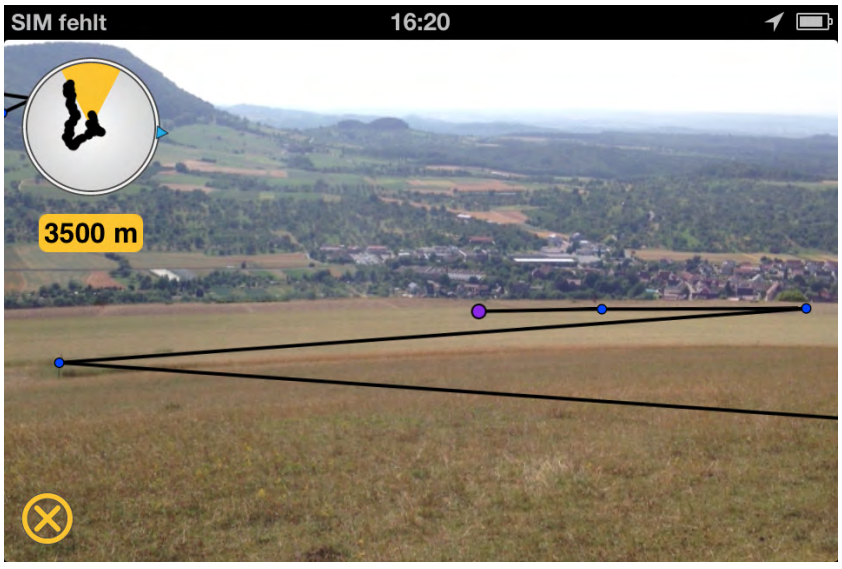


Figure 7.3.: View onto the beginning of a track

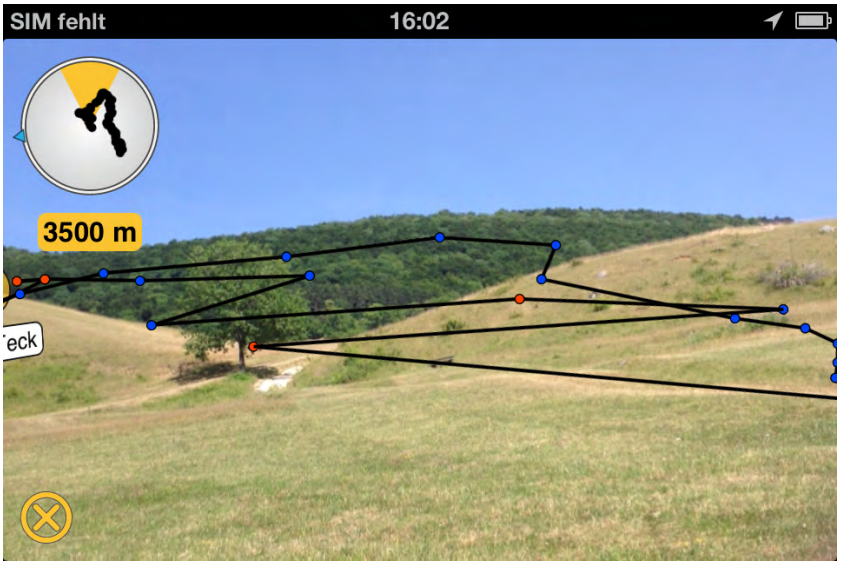


Figure 7.4.: Crossing of different track segments

the track segment on top passes on the other side of the wooded hill in front. Thus, it is painted in blue color.

Only a single track point emerges from behind the grassed hill that is situated right in front of the user in Figure 7.5. Here, again, the view is slightly shifted to the bottom. The prominent track point is thus situated on top of the small wooded hill in the back. Near the right edge of the view, a grey track point indicates a missing elevation data value.

Finally, Figure 7.6 shows the end of the track including a highlighted last track point. Here, the first part of the track passes behind a hill, while the last fraction is almost completely visible. The last but fifth track point, however, is situated in a small valley and the path reaches the target from behind the hill. This is why the last but second track point is also painted in blue.

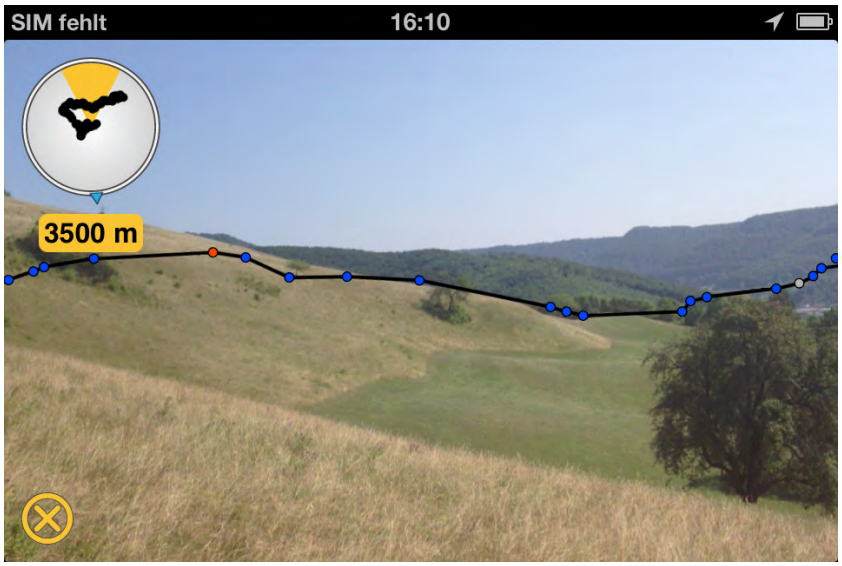


Figure 7.5.: Track behind a hill

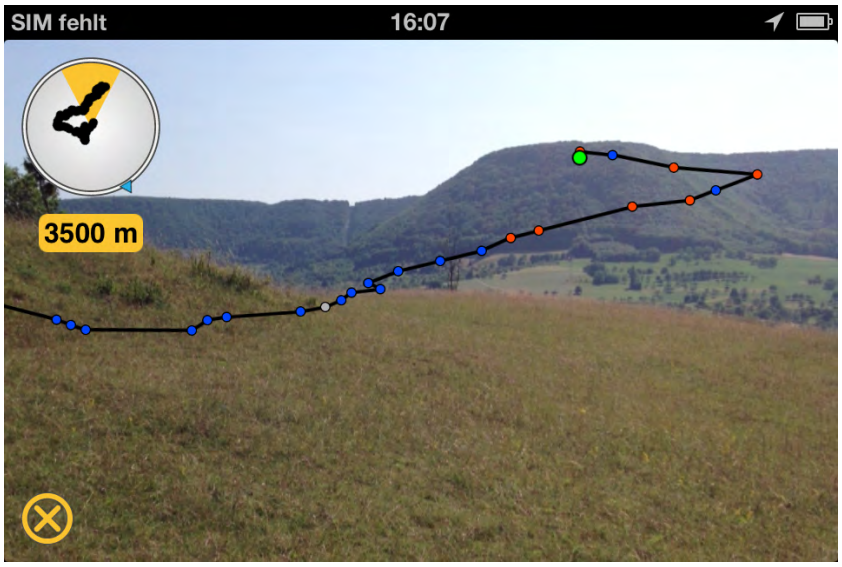


Figure 7.6.: View onto the target point of the track

8 Summary

The navigation component has been aimed to demonstrate the feasibility of an application that navigates the user alongside a track with the help of augmented reality. Thereby, the existent augmented reality engine AREA constituted the base of the navigation component's functionality. The engine displays points of interest on the camera overlay view of the iPhone. With the navigation component, it has become possible to display track points, which are distinguishable from points of interest by their smaller size and their red color. Moreover, no name is shown for a track point. Between each track point and its preceding one, a line is drawn onto the camera overlay view. Thus, the order of the track points can mostly be recognized. If a path crosses the same contour line more than once, overlapping connection lines might appear. In this case, the further route cannot clearly be traced. This also happens if segments of the displayed path are situated very close to each other. Although these complications become less in mountainous regions, they do still occur frequently. The beginning and the end of the track can, however, clearly be identified by the bigger size and different color of the corresponding track point. By increasing the size of the *LocationView* on which geolocations are drawn, connection lines to track points that are outside the view field are displayed in most cases. Still, there always remain track points outside the *LocationView*. Thus, on that way, it is not possible to consider all of them.

TrackPoint objects are stored the same way as points of interest to be displayed on the camera overlay view. So, all loaded *PointOfInterest* objects will be shown on the same screen on which the chosen track is placed. Thereby, the algorithm that calculates the track points' visibility is very efficient. This could be achieved by approximating the selection of elevation points that is compared to intermediate points alongside the user's view line. The efficiency of this algorithm is, however, combined with a slight loss in the correctness of a track point's visibility. Three different colors distinguish visible track points, not visible ones, and such for which no elevation data exists. The update of these track points concerning visibility can be performed quickly. This is caused by both, the efficiency of the visibility algorithm and the integration of this update into the general geolocation update procedure of AREA.

By pressing a button on the initial map view of AREA, the user is led to a list of tracks from which he or she can choose the one he or she wants to have displayed. On this list, a percentage value provides the user with information about how many of a track's points are covered by elevation data. Thereby, the surroundings of the track are not considered. It could, in some particular cases, occur that a great part of the area in between the user and the track point is not covered by elevation data, although there is an elevation value available for the track point's position itself. Here, the percentage would give a false impression of the real data value coverage. If a track has once be chosen to be displayed on the camera overlay view, its track points are loaded and the same track will be displayed even after choosing a different track from the list. Thus, different tracks can be shown on one and the same view. However, since they are not distinguishable, comparing becomes difficult. As the prevalent GPX-format is used to obtain *Track* objects for the navigation component, tracks from other sources can easily be added to the application. Moreover, by adding a track's points to the general *Store* of AREA, they are also displayed on the AREA radar view.

With the SRTM X-band elevation data, a suitable data source was found. Still, it only covers about 60% of the earth's surface. For testing, however, it has been suitable to choose a track that was situated within the covered area. The DTED-files containing the elevation data were read and their information was stored in a sqlite3 database. This database is accessed by the application, selecting a whole tile of elevation data and transmitting it into a dictionary. Since all elevation data is gained from the database, no internet connection is required to assure the application's functionality. Finally, a good maintainability is given through the navigation component's modular structure. However, changes, e.g., in the size of the dictionaries are difficult to perform, because no constants have been used to describe this value. Although the navigation component's source code has been commented, some parts may still be difficult to comprehend by means of the existent documentation.

In Table 8.1, the results of the application's implementation are rated according to their compliance with the requirements of the navigation component.

Table 8.1.: Compliance of the Requirements

Requirement	Rating
Track points are displayed on the camera overlay view and appear different from points of interest.	++
Information about the order of track points belonging to a certain track is displayed.	+
The track is displayed in such way that the further route can be traced clearly.	○
The beginning and the end of a track is marked.	++
Even if one track point is within the current view field, but one of its neighbors not, their connection is displayed.	+
Points of interest can be included in the camera overlay view with the displayed track.	++
Efficient algorithm to calculate the visibility of a track point.	++
Correctness of track point visibility	+
Visible and not visible parts of a track are graphically distinguishable.	++
Track points for which no elevation data exists are marked.	++
Quick update of a track point's visibility.	++
The user can choose from a list of tracks which track he or she wants to be displayed.	++
Information about the coverage of elevation data is given for all tracks.	+
Different tracks can be displayed in one camera overlay view and are graphically distinguishable.	○
Tracks from other sources can be added easily.	++
The selected track is displayed on the radar view if it is within the currently set radius.	++
A data source for elevation data is found and processed.	++
Correctness and coverage of elevation data.	+
A connection to the database is established and elevation data is gained on that way.	++
The application works without internet connection.	++
Provide maintainability.	+
Complete documentation of source code.	+

9 Outlook

The here presented version of the navigation component is still improvable and extendable in many ways.

First, some improvements can be made to achieve a better compliance with the application's requirements (cf. *8 Summary*). So, the correctness of the visibility algorithm can be increased by interpolating two adjacent elevation points in between which the great circle line crosses (cf. *4.5 Alternative Ways of Calculation*). However, the impact of this alternative algorithm upon the performance of the geolocations' update is difficult to estimate, and should thus be tested. Probably, an efficiently calculable approximation could also be found for this method. To improve a displayed track's ability to be traced by the user, track points could be numbered consecutively besides being graphically connected by lines. Moreover, if two segments of a track pass very close to each other, their visualization should be simplified by, e.g., removing the views of some of the concerned track points. To make different tracks distinguishable, when displayed on the same camera overlay view, the connection lines of their track points should be drawn in different colors. Thus, they can be compared among each other. Furthermore, constants should be introduces to hold values that might need to be modified in advance. The size of the dictionaries used to store elevation data is one example.

Next, some functional extensions could be made to further improve the application's usability. Thus, it would be expedient to extend the application's features by adding a zoom functionality. Currently, only the radius, in which geolocations are displayed, can be changed by a slider. A zoom, however, would enable the user to get a more detailed view onto a distant section of a track. Furthermore, it would be convenient to allow the user to set the size of the *LocationView* him- or herself (cf. *6.6 Drawing Connection Lines*). In the current version of the navigation component, there is no possibility to add or remove tracks from within the running application. So, a button should be added to the track list view that allows adding new tracks in form of GPX-files. With the iPhone not supporting a freely accessible file system, this could, however, be difficult to implement. Then again, it should also be possible to remove the GPX-files of available tracks from within the application. Not only the data files

of tracks should be removable, the user should likewise be able to remove a track, he or she has once chosen to be displayed from the camera overlay view. The application's efficiency could be improved, if the coverage value of a track was only calculated once and then stored into a property list. Up to now, only elevation data tiles that are situated in the northern and eastern hemisphere are available in the database. To calculate with coordinate values of the southern and western hemisphere, they need to be multiplied with -1. Generally, a higher accuracy of the application could be achieved by relying on a more detailed and correct data source. This could, for example, be given by elevation values gained from topographic maps that are provided by local land surveying offices (cf. Subsection *3.5.2 Topographic Maps*). However, it is questionable if this data is really more detailed than the SRTM X-band data.

Finally, some improvements could be attained if the device's hardware, on which the navigation component is executed, would support some additional features. So, it would slightly improve the results of the visibility algorithm, if elevation values could be gained in relation to the WGS-ellipsoid. A much more significant improvement of the application's accuracy could, however, be achieved by obtaining elevation data of a barometer sensor (cf. Section *4.3 Accuracy of Position Values*). Up to now, both of these features are, however, not supported by the current models of the iPhone series. In a nutshell, implementing the presented application on an Android device would bring the following advantages:

- Support of barometric altitude measurement (at least for some devices)
- Direct access to height values measured via GPS
- Free user access to the device's file system to add and remove GPX-files

Another possibility is given by implementing a corresponding application for a handheld navigation device that is equipped with a camera[1]. The navigation component is meant to be used in combination with other navigation tools (cf. Section *1.1 Contribution*). By integrating its functionality into a navigation system, a single device would be enough to be taken onto activities in the mountains. Moreover, the application may benefit of the high-sensitive GPS receiver and the robustness, those devices generally come with.

An augmented reality application like the here presented one could help optimizing certain processes of, e.g., warehouse management [42] [43] [44].

[1] e.g. Garmin GPSMAP 62stc [17]

A
E-Mails

A.1. Request about how to obtain WGS84 height data

Please include the line below in follow-up emails for this request.

Follow-up: 241368420

Hello Ruediger,

Thank you for contacting Apple developer Technical Supprt (DTS). Our engineers haver reviewed your request and have concluded that there is no supported way to achieve the desired functionality given the current shipping system configurations.

If you would like for Apple to consider adding support for raw GPS data in future, please submit an enhancement request via the Bug Report tool at <http://bugreport.apple.com>.

While you were initially charged a technical support incident for this support request, we have assigned a replacement incident back to your account.

Thank you for taking the time to file this report. We truly appreciate your help in discovering and isolating issues.

Best Regards,

Developer Technical Support
Apple Worldwide Developer Relations
****

THE INFORMATION CONTAINED IN THIS MESSAGE IS UNDER NON-DISCLOSURE
****

Name: Ruediger Pryss
Person ID: 1085378199
E-mail: apple@dbis.info
Team Name: Ruediger Pryss
Team ID: M255V9KULF
Team Type: Individual

PLATFORM AND VERSION
iOS
iphone, iOS 6.0

DESCRIPTION OF PROBLEM
I am developing an iPhone App in context of my thesis at Ulm University (Germany). Therefore I use a data source that consists of geographic data in WGS84. The altitudes of this data is also in relation to this reference ellipsoid without considering a geoid model (e.g. EGM96).

As far as I know, the altitude data calculated by the iPhone considers a geoid model. Is there any possibility to obtain the GPS raw data from the iPhone, which includes the elevation in relation to the ellipsoid model WGS84? If not, what other solution do you propose?

Thank you for you help!

STEPS TO REPRODUCE
—

A.2. Request about which geoid model is used

Please include the line below in follow-up emails for this request.

Follow-up: 241368420

Hello Ruediger,

Thank you for your follow-up email.

We do not provide raw GPS data or information on how GPS data works on the device. Please file an enhancement request for documentation to be made available for obtaining raw GPS data.

You may submit an enhancement request using the Bug Reporter tool at <http://bugreport.apple.com>.

Thank you for taking the time to file this report. We truly appreciate your help in discovering and isolating issues.

Best Regards,

Developer Technical Support
Apple Worldwide Developer Relations

****
THE INFORMATION CONTAINED IN THIS MESSAGE IS UNDER NON-DISCLOSURE
****

Hello,

can you tell me which geoid / quasi-geoid model is used? Is that EGM96 or a local one, depending on the region?

Best Regards,
Lisa

B
Bibliography

[1] Apple Inc. Programming with Objective-C. https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html, 2012. [Online; accessed 10 - July - 2013].

[2] EADS Astrium. Astrium, an EADS Company. http://www.astrium.eads.net/, 2013. [Online; accessed 28 - June - 2013].

[3] The Perl Community. The Perl Programming Language. http://www.perl.org/, 2013. [Online; accessed 27 - June - 2013].

[4] Adrian Covert. Why The Barometer is android's New Trump Card. http://www.gizmodo.com.au/2011/10/why-the-barometer-is-androids-new-trump-card/, 2011. [Online; accessed 7 - July - 2013].

[5] Peter H. Dana. Geodetic Datum Overview. http://www.colorado.edu/geography/gcraft/notes/datum/datum.html, 2013. [Online; accessed 06 - July - 2013].

[6] Ubiquitous Web Domain. Extensible Markup Language (XML). http://www.w3.org/XML/, 2013. [Online; accessed 28 - June - 2013].

[7] eoPortalDirectory. SRTM. https://directory.eoportal.org/web/eoportal/satellite-missions/s/srtm, 2013. [Online; accessed 21 - June - 2013].

[8] ESRI. What is a Coordinate System? http://edndoc.esri.com/arcsde/9.1/general_topics/what_coord_sys.htm, 2005. [Online; accessed 7 - July - 2013].

[9] Dan Foster. GPX schema version 1.1. http://www.topografix.com/GPX/1/1/gpx.xsd, 2004. [Online; accessed 01 - July - 2013].

[10] Dan Foster. GPX: the GPS Exchange Format. http://www.topografix.com/gpx.asp, 2013. [Online; accessed 28 - June - 2013].

[11] TU Bergakademie Freiberg. GPS - TU Bergakademie Freiberg. http://www.google.de/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0CC8QFjAA&url=http%3A%

2F%2Ftu-freiberg.de%2Ffakult3%2Ffsr3%2Fskripte%2FPositionsbestimmung.ppt&ei=9yjYUaTsD4joOqvhgLAC&usg=AFQjCNEbIBFlOfnJfZEVzFih4XHQa5z_pQ&bvm=bv.48705608,d.ZWU, 2013. [Online; accessed 6 - July - 2013].

[12] Bundesamt für Kartographie und Geodäsie. Bestimmung der Höhenbezugsfläche von Deutschland. http://www.bkg.bund.de/nn_175426/DE/Bundesamt/Geodaesie/RefSys/RefHoehe/Hoehe04__node.html__nnn=true, 2013. [Online; accessed 6 - July - 2013].

[13] Bundesamt für Kartographie und Geodäsie. Onlineberechnung von Quasigeoidhöhen mit dem GCG2011. http://www.bkg.bund.de/DE/Bundesamt/Geodaesie/GeodIS-WA/WApp/GidBer/gidber00__node.html__nnn=true, 2013. [Online; accessed 24 - Juni - 2013].

[14] Deutsches Zentrum für Luft-und Raumfahrt. FAQs. http://www.dlr.de/eoc/en/Portaldata/60/Resources/dokumente/7_sat_miss/srtm_faq_en.pdf, 2012. [Online; accessed 21 - June - 2013].

[15] Deutsches Zentrum für Luft-und Raumfahrt. Höhenmodelle der SRTM-Mission kostenfrei zur Verfügung. http://www.dlr.de/dlr/desktopdefault.aspx/tabid-10212/332_read-817/year-all/#gallery/1675, 2013. [Online; accessed 21 - June - 2013].

[16] Deutsches Zentrum für Luft-und Raumfahrt. TanDEM-X. http://www.dlr.de/rd/desktopdefault.aspx/tabid-2440/3586_read-16692/, 2013. [Online; accessed 24 - June - 2013].

[17] Garmin. GPSMAP 62stc. https://buy.garmin.com/de-DE/DE/outdoor-freizeit/handgerate/gpsmap-62stc/prod89557.html, 2013. [Online; accessed 15 - July - 2013].

[18] Garmin. Selective-Availability. http://www.garminservice.de/weitere-informationen/begriffserklaerungen/selective-availability.php, 2013. [Online; accessed 7 - July - 2013].

[19] Philip Geiger. Entwicklung einer Augmented Reality Engine am Beispiel iOS. Bachelor's thesis, Ulm University, 2012.

[20] Vivid Planet Software GmbH. Find Google Maps coordinates - fast and easy! http://www.mapcoordinates.net/en, 2013. [Online; accessed 21 - June - 2013].

[21] Google. Android. http://www.android.com/, 2013. [Online; accessed 7 - July - 2013].

[22] Google. Google Earth. http://www.google.de/intl/de/earth/index.html, 2013. [Online; accessed 27 - June - 2013].

[23] Google. SRTM4.1 - Google Earth Galerie. http://www.google.com/gadgets/directory?synd=earth&hl=de&id=834438802814, 2013. [Online; accessed 21 - June - 2013].

[24] D. Richard Hipp. About SQLite. http://www.sqlite.org/about.html, 2013. [Online; accessed 27 - June - 2013].

[25] Spot image. DIMAP. http://www2.astrium-geo.com/files/pmedia/public/r455_9_formatdimap_eng_sept2010.pdf, 2012. [Online; accessed 27 - June - 2013].

[26] Apple Inc. NSXMLParser Class Reference. https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSXMLParser_Class/

Reference/Reference.html, 2011. [Online; accessed 15 - July - 2013].

[27] Apple Inc. iOS 6. http://www.apple.com/de/ios/, 2013. [Online; accessed 10 - July - 2013].

[28] Apple Inc. iOS 7. http://www.apple.com/de/, 2013. [Online; accessed 7 - July - 2013].

[29] Apple Inc. Technische Daten des iPhone 4S. http://www.apple.com/de/iphone/iphone-4s/specs.html, 2013. [Online; accessed 10 - July - 2013].

[30] Apple Inc. Xcode. https://developer.apple.com/xcode/, 2013. [Online; accessed 10 - July - 2013].

[31] Reinhard Jung. Entity Relationship Model (ERM). http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/daten-wissen/Datenmanagement/Daten-/Entity-Relationship-Model--/, 2013. [Online; accessed 05 - July - 2013].

[32] James King. The Geometry of the Law of Cosines. http://www.math.washington.edu/~king/coursedir/m445w03/class/02-03-lawcos-answers.html, 2004. [Online; accessed 9 - July - 2013].

[33] R. Knippers. Reference surfaces for mapping. http://kartoweb.itc.nl/geometrics/Reference%20surfaces/refsurf.html, 2009. [Online; accessed 6 - July - 2013].

[34] Komoot. Komoot. http://www.komoot.de/, 2013. [Online; accessed 01 - July - 2013].

[35] Jörg Krämer and Ralf Schulte. Markscheiderische Vermessung Höhenmessung. http://www.ifm.rwth-aachen.de/cms/upload/download/Bachelor-PO10/B3V/BV3-03_MV_Hoehenmessung.pdf, 2013. [Online; accessed 7 - July - 2013].

[36] LTE. Smartphone Sensoren. http://www.lte800.com/sensoren-galaxy-s4-temperatur-barometer-feuchtigkeit.html, 2013. [Online; accessed 7 - July - 2013].

[37] Dave Mark, Jack Nutting, and Jeff LaMarch. *Beginning iOS 5 Development*. Apress, 2011.

[38] Lauren Martins. Barometric sensor on android devices, xcsoar vario update to use them please. http://www.gizmodo.com.au/2011/10/why-the-barometer-is-androids-new-trump-card/, 2011. [Online; accessed 7 - July - 2013].

[39] Sarab Najim. Geodaetische Bezugssysteme. http://www.geo.informatik.uni-bonn.de/teaching/pg_new/vortraege/Geodaetische_Bezugssysteme.pdf, 2004. [Online; accessed 06 - July - 2013].

[40] OpenLayers. Great Circle Distance. http://trac.osgeo.org/openlayers/wiki/GreatCircleAlgorithms, 2013. [Online; accessed 8 - July - 2013].

[41] Antje Pacholik and Maurice Gotthardt. Die amtlichen Bezugssysteme ETRS 89 und DHHN 92. http://www.galitzki.de/Grundl1n.pdf, 2005. [Online; accessed 6 - July - 2013].

[42] Rüdiger Pryss, David Langer, Manfred Reichert, and Alena Hallerbach. Mobile task management for medical ward rounds - the medo approach. In *1st Int'l Workshop on*

Adaptive Case Management (ACM'12), BPM'12 Workshops, number 132 in LNBIP, pages 43–54. Springer, September 2012.

[43] Rüdiger Pryss, Julian Tiedeken, Ulrich Kreher, and Manfred Reichert. Towards flexible process support on mobile devices. In *Proc. CAiSE'10 Forum - Information Systems Evolution*, number 72 in LNBIP, pages 150–165. Springer, 2010.

[44] Manfred Reichert and Barbara Weber. *Enabling Flexibility in Process-Aware Information Systems: Challenges, Methods, Technologies*. Springer, Berlin-Heidelberg, 2012.

[45] Andreas Robecke, Rüdiger Pryss, and Manfred Reichert. Dbischolar: An iphone application for performing citation analyses. In *CAiSE Forum-2011*, number Vol-73 in Proceedings of the CAiSE'11 Forum at the 23rd International Conference on Advanced Information Systems Engineering. CEUR Workshop Proceedings, June 2011.

[46] Universität Rostock. Geoid. `http://www.geoinformatik.uni-rostock.de/einzel.asp?ID=792`, 2010. [Online; accessed 6 - July - 2013].

[47] Universität Rostock. Normalhöhe. `http://www.geoinformatik.uni-rostock.de/einzel.asp?ID=-352741668`, 2010. [Online; accessed 6 - July - 2013].

[48] Universität Rostock. Quasigeoid. `http://www.geoinformatik.uni-rostock.de/einzel.asp?ID=-1986925461`, 2010. [Online; accessed 6 - July - 2013].

[49] Johannes Schobel, Marc Schickler, Rüdiger Pryss, Hans Nienhaus, and Manfred Reichert. Using vital sensors in mobile healthcare business applications: Challenges, examples, lessons learned. In *9th Int'l Conference on Web Information Systems and Technologies (WEBIST 2013), Special Session on Business Apps*, pages 509–518. 2013 SCITEPRESS, May 2013.

[50] stackoverflow. How does my iPhone get its altitude? `http://stackoverflow.com/questions/13329714/how-does-my-iphone-get-its-altitude`, 2012. [Online; accessed 7 - July - 2013].

[51] stackoverflow. Is Android's GPS altitude incorrect due to not including geoid height? `http://stackoverflow.com/questions/11168306/is-androids-gps-altitude-incorrect-due-to-not-including-geoid-height`, 2012. [Online; accessed 6 - July - 2013].

[52] Chris Veness. Calculate distance, bearing and more between Latitude/Longitude points. `http://www.movable-type.co.uk/scripts/latlong.html`, 2012. [Online; accessed 8 - July - 2013].

[53] Michaela Wagner. SRTM DTED Format. `http://www.dlr.de/caf/Portaldata/60/Resources/dokumente/7_sat_miss/SRTM-XSAR-DEM-DTED-1.1.pdf`, 2003. [Online; accessed 22 - June - 2013].

[54] Eric W. Weisstein. Great Circle. From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/GreatCircle.html`, 2013. [Online; accessed 8 - July - 2013].

[55] Wikipedia. USGS DEM — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=USGS_DEM&oldid=483715549`, 2012. [Online; accessed 27 - June - 2013].

[56] Wikipedia. Atan2 — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Atan2&oldid=558115065, 2013. [Online; accessed 8 - July - 2013].

[57] Wikipedia. Bessel-Ellipsoid — Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=Bessel-Ellipsoid&oldid=116369526, 2013. [Online; accessed 6 - July - 2013].

[58] Wikipedia. Digital elevation model — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Digital_elevation_model&oldid=554258044, 2013. [Online; accessed 22 - June - 2013].

[59] Wikipedia. DTED — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=DTED&oldid=544174191, 2013. [Online; accessed 27 - June - 2013].

[60] Wikipedia. Geodätisches datum — Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=Geod%C3%A4tisches_Datum&oldid=120035885, 2013. [Online; accessed 6 - July - 2013].

[61] Wikipedia. Geographic coordinate system — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Geographic_coordinate_system&oldid=562810643, 2013. [Online; accessed 6 - July - 2013].

[62] Wikipedia. Geoid — Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=Geoid&oldid=119529825, 2013. [Online; accessed 6 - July - 2013].

[63] Wikipedia. Model–view–controller — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93controller&oldid=560711945, 2013. [Online; accessed 26 - June - 2013].

[64] Wikipedia. OpenStreetMap — Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=OpenStreetMap&oldid=119477026, 2013. [Online; accessed 21 - June - 2013].

[65] Wikipedia. Spatial Data Transfer Standard — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Spatial_Data_Transfer_Standard&oldid=543788886, 2013. [Online; accessed 27 - June - 2013].

[66] Wikipedia. SRTM-Daten — Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=SRTM-Daten&oldid=118839766, 2013. [Online; accessed 24 - June - 2013].

[67] Ed Williams. Aviation Formulary V1.46. http://williams.best.vwh.net/avform.htm, 2011. [Online; accessed 10 - July - 2013].